Transaction Processing Facility

Application Programming

Version 4 Release 1

SH31-0132-13

Transaction Processing Facility

Application Programming

Version 4 Release 1

SH31-0132-13

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xv.

Fourteenth Edition (June 2002)

This is a major revision of, and obsoletes, SH31-0132-12 and all associated technical newsletters.

This edition applies to Version 4 Release 1 Modification Level 0 of IBM Transaction Processing Facility, program number 5748-T14, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

IBM welcomes your comments. Address your comments to:

IBM Corporation
TPF Systems Information Development
Mail Station P923
2455 South Road
Poughkeepsie, NY 12601-5400
USA

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Notices

References in this book to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service in this book is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department 830A
Mail Drop P131
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this book to non-IBM Web sites are provided for convenience only and do not in any way serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this book or accessed through an IBM Web site that is mentioned in this book.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AD/Cycle
AIX
BookManager
C/370
DB2
DFSMS/MVS
Distributed Relational Database Architecture
DRDA
IBM
Language Environment
MQSeries

MVS/ESA
MVS/XA
Open Class
OpenEdition
OS/2
OS/390
RS/6000
System/370
System/390

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

About This Book

This book describes IBM C and C++ language support for Transaction Processing Facility (TPF) application programming. It includes chapters about TPF application programming, writing TPF application programs in C, C++, and assembler, coding library functions, and debugging. Although this book is primarily directed toward application programmers, introductory and reference sections will also be of interest to TPF system programmers and customer system and middleware programmers. Tools providers will also see a need for the information in this book. It also serves as a user's guide to application programming in the TPF operating system.

For details about the C and C++ language functions referenced in this book, see the *TPF C/C++ Language Support User's Guide*. For details about the C language functions that are provided with Transmission Control Protocol/Internet Protocol (TCP/IP) support, see *TPF Transmission Control Protocol/Internet Protocol*. For details about the assembler language macros, see *TPF General Macros*.

In this book, abbreviations are often used instead of spelled-out terms. Every term is spelled out at first mention followed by the all-caps abbreviation enclosed in parentheses; for example, Systems Network Architecture (SNA). Abbreviations are defined again at various intervals throughout the book. In addition, the majority of abbreviations and their definitions are listed in the master glossary in the *TPF Library Guide*.

Who Should Read This Book

This book is intended for:

- Application programmers who already understand some general TPF programming concepts. It offers guidance on how to apply knowledge of assembler, C, and C++ languages to programming in the TPF system environment.
- Webmasters who want to use the TPF system as a Web server site. It offers guidance on how to use Internet server applications, how to port an application to the TPF system that is compliant with the Portable Operating System Interface for Computer Environments (POSIX) standards, and how to start a TPF application from the Internet.

  These users are expected to be programmers who have some familiarity with the TPF system, POSIX standards, UNIX, and the Internet.

Conventions Used in the TPF Library

The TPF library uses the following conventions:

Conventions	**Examples of Usage**
italic	Used for important words and phrases. For example: A *database* is a collection of data. Used to represent variable information. For example: Enter **ZFRST STATUS MODULE** *mod*, where *mod* is the module for which you want status.

Conventions	**Examples of Usage**
bold	Used to represent text that you type. For example: Enter **ZNALS HELP** to obtain help information for the ZNALS command. Used to represent variable information in C language. For example: **level**

`monospaced`	Used for messages and information that displays on a screen. For example:
	`PROCESSING COMPLETED`
	Used for C language functions. For example:
	`maskc`
	Used for examples. For example:
	`maskc(MASKC_ENABLE, MASKC_IO);`

bold italic	Used for emphasis. For example: You ***must*** type this command exactly as shown.
Bold underscore	Used to indicate the default in a list of options. For example: **Keyword=OPTION1 | DEFAULT**
Vertical bar |	Used to separate options in a list. (Also referred to as the OR symbol.) For example: **Keyword=Option1 | Option2** **Note:** Sometimes the vertical bar is used as a *pipe* (which allows you to pass the output of one process as input to another process). The library information will clearly explain whenever the vertical bar is used for this reason.
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord=*option*

Scale	Used to indicate the column location of input. The scale begins at column position 1. The plus sign (+) represents increments of 5 and the numerals represent increments of 10 on the scale. The first plus sign (+) represents column position 5; numeral 1 shows column position 10; numeral 2 shows column position 20 and so on. The following example shows the required text and column position for the image clear card.
	`|...+....1....+....2....+....3....+....4....+....5....+....6....+....7...`
	`LOADER   IMAGE CLEAR`
	Notes:
	1. The word LOADER must begin in column 1.
	2. The word IMAGE must begin in column 10.
	3. The word CLEAR must begin in column 16.

Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

IBM Transaction Processing Facility (TPF) 4.1 Books

- *TPF ACF/SNA Data Communications Reference*, SH31-0168
- *TPF Application Programming*, SH31-0132
- *TPF Application Requester User’s Guide*, SH31-0133
- *TPF C/C++ Language Support User’s Guide*, SH31-0121
- *TPF Concepts and Structures*, GH31-0139

- *TPF Database Reference*, SH31-0143
- *TPF Data Communications Services Reference*, SH31-0145
- *TPF General Macros*, SH31-0152
- *TPF General Information*, GH31-0147
- *TPF Library Guide*, GH31-0146
- *TPF Main Supervisor Reference*, SH31-0159
- *TPF Migration Guide: Program Update Tapes*, GH31-0187
- *TPF Operations*, SH31-0162
- *TPF Program Development Support Reference*, SH31-0164
- *TPF Programming Standards*, SH31-0165
- *TPF System Generation*, SH31-0171
- *TPF System Installation Support Reference*, SH31-0149
- *TPF System Macros*, SH31-0151
- *TPF System Performance and Measurement Reference*, SH31-0170
- *TPF Transmission Control Protocol/Internet Protocol*, SH31-0120.

IBM Systems Network Architecture Books

- *IBM Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*, SC30-3269
- *IBM Systems Network Architecture LU 6.2 Reference: Peer Protocols*, SC31-6808
- *IBM Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084.

IBM Message Queuing Books

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Clients*, GC33-1632
- *MQSeries Distributed Queue Management Guide*, SC33-1139
- *MQSeries for AIX Application Programming Reference*, SC33-1374
- *MQSeries for OS/2 Application Programming Reference*, SC33-1370
- *MQSeries for MVS/ESA Application Programming Reference*, SC33-1212
- *MQSeries Message Queue Interface Technical Reference*, SC33-0850.

IBM High-Level Language Books

- *C/C++ for MVS/ESA V3R2 Language Reference*, SC09-2150
- *C/C++ for MVS/ESA V3R2 Library Reference*, SC23-3881
- *C/C++ for MVS/ESA V3R2 Programming Guide*, SC09-2164
- *C/C++ for MVS/ESA V3R2 User's Guide*, SC09-2205
- *IBM C/370 Diagnosis Guide and Reference*, LY09-1804
- *IBM C/370 Programming Guide*, SC09-1384
- *IBM C/370 User's Guide*, SC09-1264
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663

- *OS/390 C/C++ User's Guide*, SC09-2361
- *Programming Guide SAA AD/Cycle C/370*, SC09-1841
- *Programming Guide SAA AD/Cycle Language Environment/370*, SC09-1840
- *SAA AD/Cycle C/370 Language Reference*, SC09-1762
- *SAA AD/Cycle C/370 Library Reference*, SC09-1761
- *SAA AD/Cycle C/370 User's Guide*, SC09-1763
- *SAA Common Programming Interface C Reference - Level 2*, SC09-1308.

Miscellaneous IBM Books

- *AIX Version 4.1 Commands Reference*, SBOF-1851
- *Character Data Representation Architecture Reference and Registry*, SC09-2190
- *DFSMS/MVS Version 1 Release 2 Access Method Services for VSAM Catalogs*, SC26-4905
- *ESA/370 Principles of Operation*, SA22-7200
- *ESA/390 Principles of Operation*, SA22-7201
- *Language Environment for MVS & VM Programming Reference*, SC26-3312
- *Language Environment for MVS & VM Programming Guide*, SC26-4818
- *MVS/XA VSAM Administration: Macro Instruction Reference*, GC26-4016
- *370/XA Principles of Operation*, SA22-7085.

Non-IBM Books

- *Internet Architecture Board Standard 33, Request for Comments 1350*
- *OSF DCE Application Development Reference* (1993, Prentice Hall), ISBN 0-13-643834-2
- *UNIX Network Programming* (1990, Prentice Hall) by W. Richard Stevens, ISBN 0-13-949876-1
- *UNIX Network Programming: Networking APIs: Sockets and XTI* (2nd Edition, 1997, Prentice Hall) by W. Richard Stevens, ISBN 0-13-490012-X.

Online Information

- *Messages (Online)*
- *Messages (System Error and Offline)*.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, use one of the methods that follow. Make sure you include the title and number of the book, the version of your product and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send your comments electronically, do either of the following:
  - Go to http://www.ibm.com/tpf/pubs/tpfpubs.htm.

    There you will find a link to a feedback page where you can enter and submit comments.

  - Send your comments by e-mail to tpfid@us.ibm.com
- If you prefer to send your comments by mail, address your comments to:

  IBM Corporation
  TPF Systems Information Development
  Mail Station P923
  2455 South Road
  Poughkeepsie, NY 12601-5400
  USA
- If you prefer to send your comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 +9788

Introduction to TPF

The TPF operating system is a high performance, high availability, real-time, message-driven communications system. TPF applications typically support the functions of a business where the system is in direct contact with the user's customer. That customer may have direct access to the operating system (such as through the 3614 Customer Transaction Facility), or may be serviced by a business agent, such as a bank teller or reservation agent. In either case, one of the mandates for high performance is implied: the customer is waiting.

The second mandate is the volume of message processing required. TPF applications are generally straightforward, such as credit card verification, depositing or withdrawing money on a bank account, or reserving an airline seat. However, those functions must be performed from one to several hundred times per day at each terminal, and a typical TPF system might support thousands of terminals (although some systems of only a few hundred terminals exist).

Some Fundamental Definitions

A System/370 online computing facility or central processing complex (CPC) is defined as set of central processing units (CPUs) that are packaged together to share a common main storage. The term used in TPF documentation for the set of CPUs in a CPC is *instruction-stream engines* or *I-streams*. A CPC used in a batch data processing environment normally follows a repetitive cycle of events that can be planned and timed in detail by programmers.

In a real-time TPF environment, this is seldom the case, because the sequence of operations is unpredictable. The volume and variety of messages received is such that several messages may be in the CPC at any one time. The TPF control program is used to schedule work, allocate storage, and assess priorities continuously. It permits message processing on an I-stream and component-sharing basis to maximize use of the various system resources. These resources include main and file storage, input/output components, terminal equipment, and the processing performed by each I-stream.

Language Structures and Case Guidelines

Throughout this publication there are references to C language and assembly language structures and symbols.

Note: The C language structures are in lowercase and the assembly language structures are in uppercase. This is important because often the only difference between the C language structure and the assembly language structure is the case of the name. When this occurs, we provide only the lowercase name (C language). The assembly language equivalent is always just the uppercase spelling. When there are more differences than just the case of the name, both the C language and assembly language versions are provided.

See the glossary in *TPF Library Guide* for definitions of additional TPF system and C language terms.

TPF Application Support and Environments

The original TPF system was designed to satisfy the requirements of airline reservations agents who access available flight space and record travelers' choices. TPF updates inventories, creates passenger name records, and maintains the indexes or other database support necessary to retrieve records for display or modification, if required. The agent communicates with a traveler by telephone, but accesses the necessary data using a terminal that is linked to a central computer through a communication network.

The airline reservations application still typifies the environment for which TPF provides the optimum solution. However, any data processing environment that requires remote users to access a large common database is a potential TPF user. TPF is implemented by a variety of business types, including banks, car rental companies, electronic funds transfer systems, hotel reservation systems, message switching networks, municipal government, and consumer finance companies.

Communication Protocol Support

Although applicable to a wide variety of application areas, TPF achieves its efficiency, in part, by limiting function and flexibility in favor of extremely high availability and rapid, consistent responses. The communications area supports a limited number of line protocols, as the following list indicates.

ALC	Airlines line control (high speed lines; sometimes called SABRE line control by means of the Airlines Control Interface (ALCI feature of NCP).
BSC	Binary synchronous line control
SDLC	Synchronous data link control, an SNA protocol
SLC	Synchronous link control, an airline facility.

File Storage Support

In the area of file storage support, use of both a single access technique and rigid formatting reduce operating system requirements. Other areas with concepts that contribute to the efficiency of TPF include the management of messages, working storage, and programs. The inherent complexity of a real-time system also requires development of special-purpose test procedures, test tools, and support functions.

TPF Environments

TPF and its support reside in different environments. The primary real-time environment, under control of TPF serving the application function, is often referred to as the *online* system. Most of the secondary programs and facilities, which support the online system, operate in a batch-oriented environment under control of an MVS system. This environment is referred to as the *offline* system.

TPF uses backup equipment to satisfy its high availability requirements: the backup always is ready to carry the online function in the event of failure. The offline system may also use the backup equipment in a dual role to provide the tools for application development, maintenance of programs, maintenance of data, and batch processing of applications that are not real-time (for example, financial reports and data reduction of system measurements).

Functions Performed by the TPF System

The TPF system performs the following functions:

- Control of incoming and outgoing messages; receives and transmits messages over communication lines
- Main storage and file storage management; controls allocation and release of main and file storage as requested by programs
- Queuing of work to be done; maintains lists of entries waiting for event completion or further processing
- Priority of processing; reactivates entries on a system priority basis
- Input/output control; services all input/output operations, usually upon request of the application programs
- Error checking and error recovery; identifies, logs, and resolves (where possible) all permanent and transient equipment errors
- Operator communications; communicates with the operator and provides pertinent information requested by the operator or considered necessary by TPF
- Restart and switchover; provides a means of starting active operations in a computer system or restarting operations in the standby system.

Most operating systems perform all of the preceding functions. The performance differences of each can be attributed in large part to the concepts employed to do each function. It must be assumed in designing a real-time, high-availability system that the user's business needs cannot allow the operating system to be taken down to perform maintenance functions. TPF facilities therefore are designed, and the user's applications must be designed, for database update and maintenance to be performed in real-time and still permit good response time. For the same considerations, should a system failure occur, the restart process is rapid and uncomplicated, making most restarts successful and achievable in seconds.

Messages and Entries

TPF is an operating system in which most units of work are initiated by messages. TPF is a conversational system in that a single response must be provided for each message. The term message applies to the input characters that trigger a work unit.

Entry Control Block

When TPF receives a message or entry into the system, it assigns a storage area called an *entry control block* (ECB) to that message. The term *entry* refers to the processing associated with an entry control block. Once activated, TPF processes each entry with equal priority on a first-in, first-out basis. The life of an entry is measured from the creation to the deletion of the entry control block.

An entry in TPF is analogous to a task or process in other operating systems. The ECB, which corresponds to an MVS task control block, is the primary interface between TPF and application programs. The ECB is divided into three parts:

1. The first part is used for saving such things as registers, PSW, and status. Sections are also reserved for use by application programs and for passing data between the TPF system and application programs.
2. The second part is protected from use by application programs, and is used by the system for creating and maintaining the ECB *address space* (the ECB's view of virtual storage). Applications should not attempt to modify this area.
3. The third part is used by the system to store ECB-based data and tables, such as the macro trace table, and the program nesting area.

The term *entry* should not be confused with another common TPF term: *enter*. *Enter* refers to the use of the TPF enter/back services to transfer control to another program. The *enter*ed (or called) program uses the same ECB as the calling program. The fact that both programs use the same ECB means they are part of the same entry.

In the TPF system, programs that use the enter/back services to transfer control to other programs are called E-type or ECB-controlled.

An ENTxC macro transfers processing control to another program segment. This program segment is a processing component of the original entry where the enter macro was invoked.

Each new message causes a program called OPZERO to create a new ECB. There are different OPZERO programs for different message protocols. For example, if the message obeys SNA protocol, the OPZERO appropriate for SNA messages is invoked. The application selection programs of the control program activate the initial segment (also called the root segment) of an application (usually an application input message editor). The application's message editor invokes additional application program segments to process the message. This processing generally involves retrieving information from, or adding information to, the database, and responding to the requesting terminal. During processing, the application programs will make requests of the TPF system programs for services such as input/output, file and main storage allocation, program enters and returns, and release of control.

There is no TPF facility for allocating main storage dynamically. It is fixed at system generation time. Symbolic references and TPF facilities allow common access to the data by any application, and reconfigurations without reprogramming.

Reentrant Programs

As a program attribute, *reentrant* means that a single program may be used to produce multiple outputs while maintaining the ability to accept additional input prior to producing a previous output. At any given instant, there is a unique ECB, which is referenced by an application program. To be reentrant, the application program must not modify any storage located within the bounds of the program itself. Instead, it must reference switches, indicators or counters in an application work area that is located in the ECB or owned exclusively by the entry. This means that the program may suspend, abort, initiate, or resume processing of any number of entries at any point in their processing, without ever reinitializing. Therefore, one main storage copy of a program, by referring to different ECBs, is equivalent to multiple main storage copies of the program.

An active, assembler application program's dynamic data resides in the current ECB, whose address is, by convention, stored in register 9 (R9). Several TPF macros satisfy the need for data work areas if the work area in the ECB is insufficient. Any such work area is also indirectly referred to through the ECB. Application program segments are assembled in relation to a base address of zero. At execution time, a base register is assigned to the address of the main storage block in which the program segment has been placed. The control program always uses register 8 (R8) as the base register for application program segments.

Reentrancy in ISO-C programs is handled through stack frames. A stack frame is used during the duration of a function. The frame is retained when other functions are called and reused when the called function returns. The stack frames contain the storage required by program variables, unless the variables are declared to be static. Programs that declare data static and modify the static data are said to have

″writable static″ data. To be reentrant, programs with writable static must specify the RENT parameter when they are compiled.

Therefore, although the terms *multijobbing* and *multitasking* are not used in TPF, at any one time many entries may be present in the system (hundreds, perhaps, in large systems), each handling a unique unit of work for any number of applications. Typically, an entry processes one element of input data only. Thus, in contrast to batch-processing jobs, which may run for several hours, an entry is likely to exist only for a few seconds, or even merely a few hundred milliseconds.

Batch type operations may still be handled by TPF in the real-time environment by one of two methods. They may be started by the operator keying an input message at the console, thus causing an entry to be created. A batch of input data, such as a tape file, may be handled by the entry created by operator input. This entry would have a very long life. An alternative method would be to program a monitor that would control the creation of entries, one for each input tape record. When one entry exits, the monitor can create another, thus maintaining a level of activity appropriate to available system resources.

Database Support

The main database of the TPF system is allocated at system generation time and is shared by all entries and all applications. It comprises two categories of file space. (A third category, general data sets, and a fourth, general files, are discussed in “General Data Set and General File Support” on page 273.)

The first category is analogous to a conventional multivolume data set organized for Basic Direct Access Method (BDAM). In the TPF system this organization is called *fixed file storage*. The second category is unique to TPF, and is termed *random pool file storage*. Random pool file storage (sometimes called *pools*) is similar to the areas of main working storage, because storage is dispensed and returned as needed. When an entry requests space in random pool file storage, TPF locates an available record and passes its address to the entry. The entry should record the address in a fixed file record because ultimately the system expects the requesting entry to release the random pool file record by returning the address to the TPF system. Data is stored and retrieved at the physical record level. The record size is fixed at 4096 bytes.

There is no TPF facility for allocating file storage dynamically. It is fixed at system generation time. Applications are not aware of the physical configuration of the files. Symbolic references and TPF facilities allow common access to the data by any application, and reconfigurations without reprogramming.

Program Categories

This document makes the distinction between two categories of TPF programs: ECB and non-ECB controlled programs. An application program as discussed in this document means any user-developed program designed to run online under TPF that is not a part of the TPF system itself. When used in this document, an application program always implies an ECB-controlled program as defined in the following section.

C Language in the Control Program

The TPF control program does not support C language programs. The implementation of C requires ECB virtual memory (EVM) and register connection to an ECB.

ECB-Controlled Programs

An ECB-controlled program consists of 1 or more program segments. Assembler language and TARGET(TPF) programs must fit into a standard TPF main storage block. For practical purposes ISO-C programs have no size limit. ECB-controlled programs can be permanently assigned to main storage during online execution of the TPF system, or allocated to file storage and loaded into main storage at execution.

Main storage resident ECB-controlled programs are allocated as *core resident.* Core resident programs are ECB-controlled programs that are permanently allocated to main storage because they are used so frequently.

TPF publications also refer to ECB-controlled programs as file-resident, real-time, and E-type programs.

Non-ECB Controlled Programs

The programs that constitute the main storage resident portion of the TPF system are not required to fit into standard TPF fixed size storage blocks or to use an entry control block. Although assigned to file storage for restart purposes, these programs always are resident in main storage. TPF publications sometimes call these programs C-type programs.

Offline programs run under control of either TPF or MVS to provide function in support of the online portion of the TPF system. TPF publications often refer to these programs either as S-type (system installation) or as V-type (other offline) programs.

High Performance Option

If your TPF system includes the High Performance Option (HPO) licensed feature, it is important that you have a basic understanding of its terminology and functions. You must consider certain coding issues if you are writing application programs for an HPO system. You can find additional overview information about the High Performance Option feature in the *TPF General Information* and *TPF Main Supervisor Reference*.

An installation can have several TPF systems. If these systems function completely independently of one another, each must maintain its own database of application programs and program data. The High Performance Option feature allows these TPF systems to be interconnected so that any system can access the applications and data of some or all of the others. The shared access means that data need not be replicated from system to system.

The High Performance Option feature comprises the following 2 components that can be installed separately:

- Multiple Data Base Function (MDBF) — Permits a single copy of TPF to access multiple databases.

- Loosely Coupled (LC) Facility — Allows up to 8 TPF images to access a single database; in essence, the converse of MDBF. (An image, in this sense, is at least a TPF control program, but may also include application, system, and utility programs and data.)

Combined, these components allow interconnection of TPF systems, an important capability when multiple TPF systems are required to handle high-volume business. This interconnection can be effective only if shared data is always as current as possible, which has critical implications in the writing of application programs. Later sections describe these implications in detail.

One especially important aspect of the Multiple Data Base Function is the *subsystem*. Any TPF system running with MDBF can be divided into as many as 64 subsystems, each of which contains a complete and possibly distinct set of application programs and application data. One of the subsystems, the basic subsystem, contains the control program and system utilities for all the subsystems, and may contain basic subsystem application programs as well. Furthermore, any subsystem (including the basic subsystem) can be subdivided into as many as 128 *subsystem users*. (The total number of subsystems and subsystem users cannot exceed 128.) All subsystem users in a single subsystem share application programs and may or may not share data.

TPF Programming Conventions

TPF has established certain programming conventions to achieve high performance and ease program maintenance. It is important that the application conform to these conventions. TPF enforces some of the conventions; only good programming discipline can maintain others. Some of these conventions have been mentioned already and some will be discussed, in detail, in later sections.

Common Programming Conventions

These conventions apply to TPF applications programming in general. Those particular to assembler language and C language follow this section.

- All application programs must be reentrant. They must not contain any internal switches, indicators, or counters.
- Do not modify the fields in the ECB that TPF maintains. These fields include, but are not limited to, core block reference words, the TPF register save area, and other control information. Refer to these fields as necessary.
- Remember that TPF does not necessarily clear or initialize any working storage blocks before assigning them to the application program.

  All working storage blocks begin on a doubleword boundary.
- Symbolic references must be maintained in all areas. This will enable programming to be configuration independent and system parameters to be changed without extensive programming effort.
  - In assembly language no absolute address constants can be used; the technique of subtracting one label from another in order to obtain an actual byte count of the distance between the two locations can be used, provided the two labels are defined within the same program segment or data record.
  - Also in assembly language the names for all storage locations, registers, and I/O devices should be symbolic.
  - Use of absolute values of all types should be minimized. In C language only put absolute values in headers and associate symbolic names with these values.

  - Use the `TPF_regs` structure in C language for passing data in registers to programs where registers must be specified.
- When your application programs will operate on any I-stream of a multiengined CPC, follow the rules of multiprogramming and multiprocessing as described in the *ESA/390 Principles of Operation*. In general, TPF will handle the serial access to data records via the normal system services (for example, `getcc`, `relcc`, `find_record` or FINDC, `file_record` or FILEC, `fiwhc`, `filuc`). However, when your application interacts with a shared resource, such as a common global field, use appropriate multiprocessing techniques to access and update these areas.
- Ensure that your application programs return control to TPF within 500 milliseconds of activation. Control may be returned in several ways, some of which follow:
  - Call the `waitc` function while waiting for I/O activity to complete. This is required for any find request. A program may call the `waitc` function explicitly, or may call another function (for example, a `fiwhc` function) that, in turn, performs a wait. Note that `waitc` returns control to the application program, and resets the 500-millisecond timer, only after all I/O activity for the ECB actually has completed.
  - Call the `dlayc` or `defrc` function partway through the program logic to place the program's entry at a lower priority than other work. You might use this method if the program does little I/O activity, but might run slowly enough to risk breaking the 500-millisecond limit.
  - Call the `exit` function to end processing of the program.

  Most application programs are transaction- and I/O-oriented, so they return control to TPF frequently with each `waitc`. The timing constraint usually only becomes a concern if the programs will be retrieving much of their data from the virtual file access area. Such retrieval does not reset the 500-millisecond timeout counter. See "Virtual File Access Facility" on page 279 for more information.
- Ensure that all application programs call a `waitc` function, either explicitly or implicitly, after a read or write request to ensure that the I/O is complete before attempting to use the requested resource (for example, the file record to be read with a Find function. See "Summary of File Reference Functions and Macros" on page 260).
- Application programs should return any system resources not in use or imminently planned for use. This is an important discipline. All system resources (such as main storage blocks and file pool addresses) should be returned promptly. Because many ECBs operate concurrently, failure to do this could be a significant drain on system performance and require excessive resource allocation.
- During execution of a TPF API function, a variety of exceptional conditions may be detected. Application programs regain control only on I/O hardware errors or I/O-associated unusual conditions related to a unique entry. A function parameter error results in a forced exit and a dump. TPF takes a dump and sends an operator message only on I/O hardware errors. These conditions are recorded in the ECB on return to the application program from the TPF API function.
- Use the find-and-hold functions to hold all file storage records before you update them, to ensure the proper sequence of update.

  See "Record Hold Facility" on page 259 for more explanation. Ensure that your program holds only one record at any given time, to prevent a *lockout* condition from occurring.

Assembler Language Programming Conventions

Below is a summary listing of assembler language programming conventions:

- Except where otherwise noted, only 31-bit addresses must be passed to TPF system services.
- All assembly language application programs must fit into a 4KB block to adhere to the basic file structure and main storage management facilities of the system. Approaching the block size limit too closely, for example, within 20%, increases the difficulty of making corrections or modifications to the program.
- Application programs should not execute privileged instructions, nor explicitly issue the supervisor call (SVC) instruction.
- Application programs should not make use of the restricted-use control program macros described in *TPF System Macros*. Only the general use control program macros and the general purpose application macros are appropriate for application programs, because only these macros are guaranteed to be compatible with later releases of TPF.
- The general registers 0 through 7 are reserved for use by application programs. Their contents are saved across all control program macros. Registers 14 and 15 may also be used by application programs, but are often used by the control program to pass information between programs. The contents are not guaranteed across macro calls unless otherwise indicated.
- General register 8 always contains the base address of the active application program. It must not be altered by application programs.
- General register 9 always contains the address of the current active entry control block. It must not be altered by application programs.
- General register 10 is reserved for CP use, but can be used as a scratch register (the contents are not guaranteed across macro calls).
- General registers 11 and 12 are reserved for control program use. Their contents must not be modified by an application program.
- General register 13 is reserved for CP use, but can be used as a scratch register (the contents are not guaranteed across macro calls).
- All storage is assigned by the control program. You do not, in general, code define storage (DS) commands in your own application program to describe storage block formats. Instead, your program requests storage blocks from the control program and uses TPF or user data macros to describe the formats.
- An application macro may generate inline code or an SVC through the use of a control program macro. The SVC machine instruction is never directly used by an application.

C Language Programming Conventions

The following summary lists conventions that apply to using the C language.

- Allocate your program as 31-bit mode only. In general, TPF programs may be allocated in either 24-bit mode or 31-bit mode; however, TPF C language support requires programs in C to be in 31-bit mode.
- Give a unique 4-character name to each application program segment. For ISO-C, each application program segment must contain either a `main` function or a function with the same name as the program segment. For TARGET(TPF) programs, the name of the first external function will be used as the name of the segment.

  If you prefer to use a longer name for readability, code a `#pragma map` statement for the segment name.

  `#pragma linkage`(name ,TPF, type) is not supported for ISO-C

- Use the `system` function to activate a C program that contains a `main` function.
- Follow the register conventions established in the *TPF Program Development Support Reference* when coding programs that interact between the assembler and C language support.

TPF Advanced Program-to-Program Communications

The function provided by the TPF Advanced Program-to-Program Communications (TPF/APPC) interface is an implementation of the IBM Advanced Program-to-Program Communications (APPC) architecture. TPF/APPC is an interface that allows TPF transaction programs to communicate with remote SNA nodes that have implemented the APPC interface using LU 6.2 protocols. This application interface is provided for application transaction programs written in assembler language or in C language.

Note: Mapped conversation support is provided only through the C language interface.

The TPPCC macro, described in *TPF General Macros*, provides the assembler language interface for basic conversations. The `tppc_` C function calls provide the C language interface for basic conversations, and the `cmxxxx` C function calls provide the C language interface for mapped conversations. All TPF/APPC C function calls are described in the *TPF C/C++ Language Support User's Guide*. The CNOSC macro, described in *TPF General Macros*, provides the communication interface between the control operator and the TPF application programs.

A conversation between transaction programs is identified and controlled by a conversation control block (CCB), which is defined by data macro ICCB. A session is controlled by a session control block (SCB), which is defined by data macro ISCB. Local TPF transaction programs that are activated by remote transaction programs or by the `tppc_activate_on_receipt` or `tppc_activate_on_confirmation` function must be defined to TPF in the transaction program name table (TPNT).

This publication discusses TPF application programming in a traditional environment of host and terminals. For more information about application programming in a distributed processing environment using LU 6.2 protocols (TPF/APPC), see the *TPF ACF/SNA Data Communications Reference*.

TPF Application Requester

The TPF Application Requester (TPFAR) feature permits you to share data between a remote, DRDA level-1 compliant relational database and a TPF application using the common structured query language (SQL) interface. An LU6.2 connection is used to connect to a remote system. With TPFAR, a TPF application can directly access and retrieve or update the information residing on the remote system. TPF applications can be written in System/370 assembler language or the C language.

TPFAR uses the Distributed Data Management (DDM) architecture that allows an application program to work on data residing in a remote system. Built on top of DDM is the Distributed Relational Database Architecture (DRDA) which contains protocols for communication between relational databases.

For additional information, see *TPF Application Requester User's Guide*.

Message Queue Interface (MQI) Client

The MQI client provides TPF applications access to the standard message queue interface (MQI), supported by other MQSeries product offerings. The MQI client implements an MQSeries client on the TPF system. With this function, TPF applications can now interact with MQSeries applications running on several different platforms using a messaging and queuing model for communication services. An ISO-C interface is provided for the 11 functions that make up the MQI. The MQI client function is implemented as an ISO-C library, with the 11 MQI functions implemented as external functions in this library. A complete description of the MQI is contained in the *MQSeries Message Queue Interface Technical Reference*. Specifics about MQI clients is found in *MQSeries Clients* and *MQSeries Distributed Queue Management Guide*.

The TPF MQI client implementation is based on the standard MQSeries client server interface used by other MQSeries product offerings. MQSeries clients use MQI channels to communicate with MQ seriess. The MQI client uses LU 6.2 sessions through Common Programming Interface for Communications (CPI-C) or Transmission Control Protocol/Internet Protocol (TCP/IP) to connect with remote MQI queue managers that are capable of running MQ series function. A channel definition must be created at both the MQI client and server ends of the connection. The MQI channel directory is where the TPF system maintains a maximum of 50 channel definitions. The ZMQID ALTER, ZMQID DEFINE, ZMQID DELETE, and ZMQID DISPLAY commands are used to maintain the MQI channel directory. For more information about maintaining the MQI channel directory, see *TPF Operations*.

A TPF Transaction Example

This section presents an example transaction that is followed through the system. Some of the coding details have been simplified for the purposes of discussion. Many of the C functions used in coding this example are further discussed in “TPF Application Program Interface Functions” on page 227.

At the outset, assume a TPF system that is idle, except that the communications control program (CCP) is monitoring the lines for input. System activity is triggered when any user enters a message at a terminal. A given terminal may have access to any application in the network, but until some association is established, TPF has no knowledge of the intended destination of the message. The first message will be a request to connect the terminal to some application. This is done with a login message specifying the application. For example:

```
LOGI CRED
```

This message requests TPF to connect the inputting terminal to a credit verification application identified as CRED. (CRED is defined as a non-SNA application.) This new message would normally be placed at the bottom of a queue called the input list. In our example the system is idle so that it is processed immediately. The process flow is as follows:

Step 1 The CCP gives control to a system program called OPZERO, which performs 2 principal functions.

- OPZERO allocates an ECB from the ECB area and initializes it as an ECB. This block will be associated with this message (entry) throughout its life in the system. The ECB will be passed to all processing programs. Later, this section discusses in detail the function and format of the ECB because it is the key facility by which the application interfaces with the TPF system and controls its processing.

- OPZERO gets a main storage block into which it formats the input message and stores the address of the message block in the ECB at data level 0. The ECB contains references to 16 data levels (level 0–F). The data levels are a series of slots used to store pointers and control information for data blocks. No priority or nesting is implied by the use of the term level. All levels may be used to suit the needs of the application. However, to simplify interfaces, certain conventions have been established about the use of specific levels. One of these conventions is that the input message is passed to the application on data level zero.

Step 2 OPZERO passes control to the communications source processor (COMM SOURCE), whose principal function is to determine the intended destination of the message and to create the routing control parameter list (RCPL). The RCPL, like the ECB, will be associated with this entry throughout its life in the system; it defines the origin and destination of the message as well as descriptive information to facilitate routing and processing. COMM SOURCE stores the RCPL in the ECB work area at location EBW000.

COMM SOURCE also retrieves a routing control block (RCB) or agents assembly area (AAA), or both, depending on what the application expects. (See “Routing Control Block” on page 48) By convention, the RCB address is placed in data level 3 of the ECB, while the AAA address is placed in data level 1. In a central processing complex with more than 1 instruction stream COMM SOURCE also may be used to determine the routing of the message to an application specific I-stream or to the I-stream determined by the system. This is done by a COMM SOURCE user exit. Finally, COMM SOURCE passes control to the log processor (assuming the terminal is not already logged to an application).

Step 3 The log processor in effect connects the inputting terminal to application CRED. The log processor updates the slot for the terminal in the WGTA table. If the terminal is an SNA logical unit, the resource vector table (RVT) also is updated. The index value indicates the connection and also points to the program to be activated to begin processing the message for application CRED.

The log processor generates a message stating that the terminal has been successfully connected to application CRED and requests the TPF message router to send it to the requesting terminal. The details of how this is done are deferred to the next message.

Step 4 The terminal operator is now in a position to enter messages for the credit verification application. Until the connection is terminated all input from this terminal will be so routed. The operator enters a message requesting credit against a specific account number for a value of $75. For example:

```
/4247852601709 $75
```

The first character (/) is an action code identifying the format and expected function of the message. Now assume for clarity in following the flow that the terminal is connected to a BSC line.

Processing will begin as described in Step 1. It continues with Step 2 in creating the RCPL and retrieving the RCB. However, when COMM SOURCE looks at the WGTA slot for this terminal it finds that it is already connected to an application. The index value in the WGTA points to information in the application name table and the routing control application table, which supply the file address of the input message editor for application CRED. COMM SOURCE passes control to this program, which we will call CRC1. Figure 1 shows the conditions when CRC1 (the application) is activated.

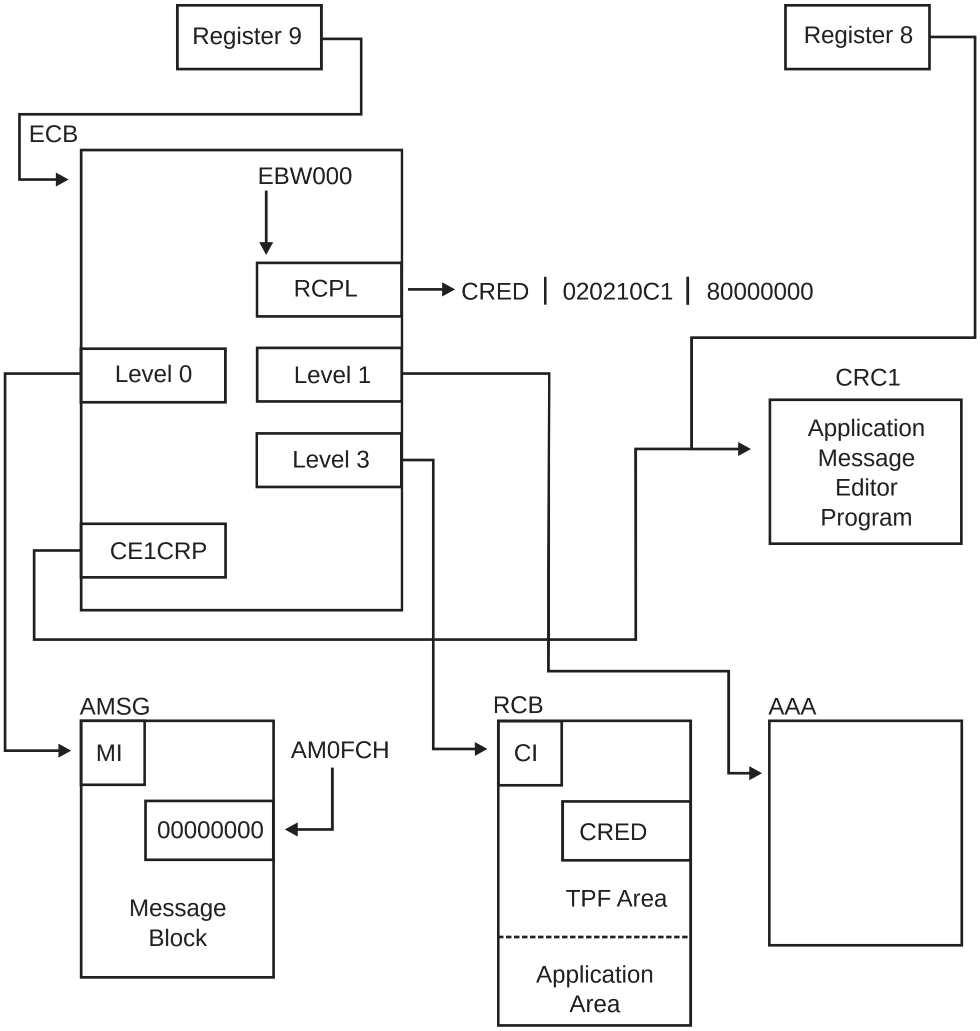


Figure 1. Conditions When Application Is Activated. The RCB and/or the AAA may be present.

To summarize:

- The ECB is in main storage, pointed to by register 9.
- The input message, formatted according to the `am0sg` structure, is in main storage, pointed to by data level 0 of the ECB.
- The routing control block (RCB) is in main storage, pointed to by data level 3 of the ECB, and it is in hold status. Or,

  - The agents assembly area (AAA) is in main storage, pointed to by data level 1 of the ECB.
  - Both an RCB and an AAA are present in main storage as described above.
- The RCPL, which specifies the origin of the message as the terminal address (LNIATA/CPUID) and the destination as CRED, is stored in the ECB at location EBW000.
- The object code for CRC1 is in main storage, pointed to by register 8.

Should this message come in from an SNA network, conditions would be identical except that there would be no RCB on level 3 (unless the application is RCB-dependent), and the origin field of the RCPL would be in SNA format (sequence number, resource ID/CPU ID).

From this point, the processing flow is exclusively in the hands of the application and unlimited variations are possible. It depends on the complexity of the functions to be performed and the application design. Conceivably, a simple application might be contained in one program segment, which would do something with the data blocks passed to it by COMM SOURCE, send its response message, and exit. On the other hand, a complex application may involve hundreds of program segments interacting with each other and TPF, extensive file accessing and main storage requirements. For the purpose of this example, assume a simple credit verification application of two program segments and arbitrarily select some commonly used functions that illustrate the manner in which application programs interface with TPF. Figure 2 on page 16 shows the logic flow for CRC1 and Figure 3 on page 20 shows the logic flow for CRC2.

Step 5 There are different procedures for assembly language programs and C language programs.

For assembly language, after first determining that this is a normal new input message, CRC1 must edit the message and determine which function is being requested. TPF provides a data macro that defines the fields in the input message and allows the program to refer to it by symbolic tags. The data macro name is AM0SG.

Example:

```
AM0SG REG=R2
```

By issuing the data macro statement, CRC1 indicates to the assembler that it intends to address the AM0SG record using register 2 as a base. The symbolic tags can be referred to by loading R2 with the pointer to the input message from level 0 of the ECB.

The input message tokenization support macro BPKDC might be used, or the program could supply its own logic for the edit. In any case, CRC1 determines that the message is a valid standard request for credit.

For C language, after first determining that this is a normal new input message, CRC1 must edit the message and determine which function is being requested. CRC1 uses the `scanf` function to read the data in. The `scanf` function locates and then parses the input message into an account number and an amount and returns the number of fields that were successfully converted and assigned. Note that the `gets` function must be implemented for the `scanf` function to work properly. The `scanf` and `gets` functions are not implemented according to the ANSI standard. See the *TPF C/C++ Language Support User's Guide* for details on `scanf` and `gets`.

Alternatively, the C language parser (`IPRSE_parse`) can be used in this example.

Example:

```
rc = scanf("/%[0123456789] $%d", acctnbr, &amount);
```

For the purposes of this discussion, assume that the message is a valid standard request for credit.

Step 6 The program then retrieves the negative credit file, which resides on the fixed file. The fixed file is a DASD file area assigned permanently to specific functions or record types, as opposed to the dynamic pools, which may be used repeatedly for various purposes. There are many types of records on the fixed file, but the TPF file access routines see it as one file. Therefore, before requesting TPF to retrieve the negative credit file record you must supply its ordinal number (relative record number) in the entire fixed file. The application only knows the ordinal number in the negative credit file. To resolve this, all records on the fixed file are assigned a record type, identified by a symbolic name.

TPF provides an index table and utility program called FACS, which, when given the symbolic record type and ordinal number ***in that type***, will calculate the ordinal number ***in the entire fixed file***. Hereafter, ordinal number in the entire fixed file will be referred to in this publication as the *symbolic file address*.

Note: An older form of this program, called FACE, operates like FACS except that a numeric rather than symbolic record type is passed to FACE. The numeric record type is set up using an equate in assembly language or a `#define` in C language; however, any changes to the FACE index table may require programs that use FACE to be recompiled. Programs that use FACS are not affected by changes to the FACS index table. See "FACE, FACS, and FAC8C" on page 251 for further information.

Assembly program CRC1 calculates the ordinal number of the desired negative credit file record and transfers control to FACS, supplying this ordinal number, the symbolic record type (for example: #VD1VD), and the location in the file address reference word (Step 8) at which the symbolic file address is to be stored.

Example:

```
ENTRC FACS
```

```
Pseudo logic for credit application segment CRC1.
-------------------------------------------------

Edit input message into acct# and amount.

Compute Negative file record address on level D2.   /* FACS or face_facs */
Retrieve Negative file record on level D2.          /* finwc or find_record */

If acct# is in Negative file record,
    Indicate negative response.
    Unhold RCB.                                     /* unfrc */
Else,
    Get floor limit address                         /*GLOBZ or glob */
    If floor limit is exceeded,
        Indicate floor limit reject.
        Unhold RCB.                                 /* unfrc */
    Else,
        Release message block (D0).                 /* crusa */
        Release Negative file record block (D2).
        Release RCB (D3).
        Unhold RCB.                                 /* unfrc */

        Call CRC2.                                  /* Process activity file */

Get block for output message on level D6.           /* getcc */
Edit output message in block on level D6.
Build output message RCPL.
Request send output message.                        /* routc */

Exit.
```

Figure 2. Logic Flow for Program CRC1

Program CRC1 calculates the ordinal number of the desired negative credit file record and calls FACS, supplying this ordinal number, the symbolic record type (for example: #VD1VD), and the location in the file address reference word (Step 8) at which the symbolic file address is stored.

In assembler, use the ENTRC macro to call FACS:

```
ENTRC FACS
```

In C, include the header file <`tpfio.h`> to declare the FACS function. FACS takes a single parameter, a pointer to a `TPF_regs` structure (defined in <`tpfregs.h`>). The fields of the `TPF_regs` structure (R0, R6, and R7) imitate the assembler interface to FACS.

The TPF C library also provides a `face_facs` function, which is a more intuitive interface for calculating a fixed file address.

Example:

```
#include <tpfio.h>
#define VD1RI "#VD1VD  "
#define VD1RI_ordinal 10
 ⋮
unsigned long rtnord ;               /* return ordinal */
 ⋮

/*
**    Locate negative credit record (VD1VD) for account on fixed file.
*/
```

```
int ffrc = face_facs(VD1RI_ordinal,VD1RI,0,D2,&rtnord) ;
                                    /* Calc fixed file address */
```

Step 7 The assembler program FACS or the ISO-C `face_facs` function calculates the symbolic file address of the desired negative credit file record, stores it at the location specified, and returns control to CRC1.

Step 8 CRC1 now can request that TPF read the record from file storage into main storage. This is done with the FINDC macro (assembly language) or the `find_record` (C language) file reference function. Before the function is called, a data level must be chosen and the FARW set up. The FARW is an 8-byte control field for file activity associated with each ECB data level. The low order 4 bytes will contain the symbolic file address because FACS or `face_facs` placed it there in Step 6.

The high order 4 bytes of the FARW are for data integrity. Normally, the requesting program must specify the record identification as an argument in the function call so that it will be entered in to the high-order 2 bytes of the FARW. The record identification will also appear in the header of the record in file storage. (This record identification is assigned to your application by an application or data base designer.) CRC1 enters the ID (for example, VD) and then requests TPF to read the record into level 2 and return control to CRC1 only after the I/O is complete. The coding for this form of the macro is:

```
FINWC D2,ERROR1
```

ERROR1 is the symbolic tag in CRC1 to which control will be returned if there is any abnormal I/O condition.

For C, the `find_record` function call requires the following arguments:

`level`
The data level of an available FARW and CBRW for use by the system.

`address`
A pointer to the file address of the record to be retrieved; `NULL` if the FARW has been preinitialized.

ID A 2-character string that must match the identification characters in the record to be retrieved; `RECIDRESET` if the identification field has been preinitialized. If the requested ID does not match the record's identification characters, the ECB error fields will indicate an identification check failure and control will return to the application at I/O completion. No comparison will be made if the identification field is zero (`'\0'`).

`rcc`
An unsigned character that must match the record code check (rcc) byte in the record to be retrieved. If the requested rcc does not match the record's rcc, then the ECB error fields will indicate an identification check failure and control will return to the application as described above for the ID argument. If rcc is zero (`'\0'`), then no comparison will be made.

type
What the record's hold status will be after I/O completion, either `NOHOLD` or `HOLD`.

CRC1 requests TPF to read the record into level 2 and return control to CRC1 only after the I/O is complete. An example of the coding for this form of the function is:

```
vd1ptr = find_record(D2, NULL, "VD", '\0', NOHOLD);
```

Step 9 TPF obtains an appropriate storage block and reads the negative credit file record into main storage, then returns control to CRC1. Assume that there are no errors or integrity failures.

Step 10 CRC1 searches the record for the input account number. Assume it is not found; the account is, therefore, in good standing, and the next step proceeds.

Step 11 The dollar value requested in the message must now be checked against a floor limit; requests greater than the floor limit are rejected. The floor limit is only a 4-byte field, but must be accessed by every entry (except those found in the negative credit file). For performance considerations, it should be stored in an area accessible to all entries but not requiring a file retrieval. For such requirements TPF maintains the global area.

The *global area* is a section of main storage accessible to all users by executing the appropriate functions. It can be used for any type of data requiring quick access, from miscellaneous constants to main storage resident data records.

By issuing one macro (GLOBZ), the assembler application program can load the base register and have access to any field in the first 4096 bytes of the global area. These 4096 byte global areas are unique to each I-stream. If there are any shared resources in these areas, provision must be made to handle the synchronization of accessing/updating these resources. It is, therefore, a common practice to place, in the first 4096 bytes, address pointers to the rest of the global area.

Example:

```
GLOBZ REGR=R5
```

The system loads register 5 with the base of the global area. By convention all tags in the global area (used by assembler language programs) begin with the character @. For example, the floor limit might be stored in a field labeled `_cflth`. CRC1 now compares the input value with the global field; it finds the request value below the floor limit and proceeds to the next step.

The `c$globz.h` C language header file contains symbolic tags and displacements for miscellaneous data fields and pointers to records in the global area. By calling one function (`glob`), the application program can examine any global field or record.

By TPF convention, all tags in the global area begin with the character "_" for C language code; however, it is important to note that when the same tags are referenced in assembler language code, the corresponding names must begin with the "@" character.

Also, other characters in the tag name are converted to lowercase. For example, the floor limit might be stored in a field labeled `_cfltn` for C language access and `atcfltn` for assembler access.

CRC1 now compares the input value with the global field; it finds the request value below the floor limit and proceeds to the next step.

Example:

```
cfltn = glob(_cfltn);            /* Check against global limit      */
if(amount > *cfltn)
  msgnexit("OVER FLOOR LIMIT\n");
```

Step 12 The next functional requirement is to check the requesting account number for excessive activity on this date and during this week.

For this purpose an activity file is maintained. All processing cannot be contained in one segment so that activity file processing will be handled by program segment CRC2.

Before passing control to CRC2, however, CRC1 will return any system resources not in use or imminently planned for use. All system resources such as main storage blocks and file pool addresses should be returned promptly. Since many ECBs operate concurrently, failure to do this could be a significant drain on system performance and require excessive resource allocation.

CRC1 finds three items no longer required:

- The input message block—the message has been edited and the account number and dollar value stored.
- The negative credit file—it has already been searched.
- The RCB—it was passed to CRC1 or CRC1 on activation, in hold status, but the application design does not require it for update or terminal lock-out. (See "Routing Control Block" on page 48.)

These blocks are returned to the system with the `crusa` function, which requires only the number of levels to be released and the names of the data levels themselves.

Because CRC1 expects to be reactivated after activity file processing, it passes control to CRC2 with the return option.

Example:

```
ENTRC CRC2
```

Figure 3 on page 20 shows the logic flow for CRC2.

C language example:

```
/*
** Discard input message, VD1VD record block, and RCB
*/

crusa(3, D2, D3, D0);         /* Release core blocks        */
unfrc(D3);                        /* Unhold the RCB file record */
```

```
Pseudo logic for credit application segment CRC2.
--------------------------------------------------

Compute Activity file record address on level D4.   /* FACS or face_facs */
Retrieve and HOLD Activity file record on level D4. /* fiwhc or find_record */

If acct# is in Activity file record,
    Update usage counts.
Else,
    If there is room for acct# in Activity record,
        Add acct# and set usage count to 1.
    Else,
        Get block for chained Activity record (D5).
        Get file address for chained record (D5).   /* getfc */
        Chain new file address to prime record.
        Initialize new Activity file record.
        Add acct# and set usage count to 1.
        File chained record (D5).                   /* filec */

File and UNHOLD prime record (D4).                  /* filuc or file_record */

Create CRC3 to record transaction log.              /* cremc */
Return to caller.
```

Figure 3. Logic Flow for Program CRC2

Step 13 The next functional requirement is to check the requesting account number for excessive activity on this date and during this week. For this purpose an activity file is maintained. The activity file also resides on the fixed file. CRC2 calculates the ordinal number of the required activity record and calls FACS (or `face_facs` for C language), passing the ordinal number, the activity file record type (for example, "`#VU1VU `"), and the level 4 FARW as the location for storing the symbolic file address.

C language example:

```
#define vu1vu "#VU1VU  "
#define vu1vu_ordinal 100  struct TPF_regs regs ;
 ⋮
/*
** Check if there has been excessive activity for this account.
** Locate credit activity record on fixed file.
*/

regs.r0 = vu1vu_ordinal                  /* Ordinal number               */
regs.r6 = (long) vu1vu;             /* Record identification (type) */
regs.r7 = (long) &(ecbptr()->ce1fa4);      /* Data level      */
FACS(&regs);                        /* Calculate record address     */
```

Step 14 The FACS macro (or the `face_facs` function) calculates the symbolic file address of the activity record, stores it in the FARW of level 4, then returns control to CRC2.

Step 15 CRC2 enters the record identification for the activity file in the FARW, then calls a find-wait-and-hold form of the `find_record` function to read the record into data level 4.

For example:

```
FIWHC D4,ERROR2
```

C language example:

```
if(!(vu1ptr = find_record(D4, NULL, "VU", '\0', HOLD)))
  serrc_op(SERRC_EXIT,00x1238, "ACTIVITY FILE FIND ERROR",NULL);
                         /* If no prime exists, error.       */
```

To assure data integrity and proper sequencing of updates when modifying a file, programs intending to update a record must ensure that the record is not currently being updated by another program. The program does this either by:

- Holding the record itself
- Holding the first record in a chain that includes the record to be updated.

  A chain is a series of records, each with an address pointer to the next record in the series. The first record in the chain is usually called the prime record. This technique will not work if the record to be updated is part of more than one chain. See "Record Hold Facility" on page 259 for further details.

The record is read successfully and searched for the input account number, but the account number is not found. This is a normal condition. Account numbers appear in this file only if used in the last seven days. An item appearing for the first time in the last seven days must be added to the file.

Step 16 CRC2 must add the item to the file and note its use today. However, the record is full and no records are chained from it. A new record must be created, chained to the first record (now called the prime record), and filed in the random pool. Unlike the fixed file, the random pool is a section of DASD storage assigned to no specific record type. It is a dynamic pool of file blocks from which any entry may request a record from TPF and return it when no longer required. The pools are divided only by record size and usage classification. (Refer to "Random Pool File Area" on page 254).

To create the new file record CRC2 takes the following steps:

1. It requests from TPF a file address from the random pool and a main storage block in which to build the record. The GETFC macro in assembly language or the `getfc` function in C language obtains the file pool address and storage block. Although transparent to the application, this record identification is an index into the RIAT and tells TPF the pool type from which to take the address. The application need know only the record identification of the it is building. The size of the acquired working storage block is determined by the attributes associated with the record identification.

   The ID ("VU" in this example) is stored in the header of the new block on level 5 and in the FARW for level 5. The ID is the same because this block is a chain extension of the prime record, whose identification is VU. All records chained together have the same record identification. (The record identification is assigned to the application by an application designer.)

   Assembly language example:

   ```
   GETCC D5,L2     Request a 1055-byte (L2) block on level 5
   ```

   C language example:

   ```
     vu1ptr->vu1fch = getfc(D5, GETFCOVERF, "VU", GETFCBLOCK,
       GETFCSERRC);

   /* model:  getfc(level,type,ID,block,error)
   ```

   The c language arguments and descriptions are as follows:

level
The data level.

type
The pool type and size to be used with the ID argument.

ID A pointer to a 2-character record identification that will be used in scanning the record identification attribute table (RIAT).

block
Whether or not a block of working storage should simultaneously be obtained.

error
Whether or not control should be returned to CRC1 in the event of an error.

2. CRC2 store upon return the new random pool address returned by TPF in the FARW for level 5 in the forward chain field of the prime record on level 4.
3. CRC2 initializes the chain record and enters the new data item and item count of 1 in the new record.
4. CRC2 files the newly created chain record and then files and unholds the prime record.

Assembly language examples:

```
FILEC D5     Files chain record from level 5.

FILUC D4     Files and unholds prime record from level 4.
```

C language example:

```
  if (newpool)                /* If we got a pool,         */
      holdstatus = NOHOLD;    /* D4 record is not locked   */
                                  /* File the D4 record block. */
  file_record (D4, NULL, NULL, '\0', holdstatus);
  if (newpool)                /* If we got a pool,         */
  file_record (D5, NULL, NULL, '\0', UNHOLD);
                                  /* file the prime record.    */
```

Step 17 The user wishes to keep a transaction log containing a copy of the input values (account number and dollar value) for each request for which credit is granted. The items are to be blocked in a 1055-byte main storage block and then written to the real-time tape (RTA tape). However, the user does not want to keep the customer and operator waiting on line for this processing, since all key decisions that relate to approving the credit have been made. CRC2, therefore, creates a separate entry to process the transaction log independently, passing to it the required log data and specifying the program to be called for initial processing (CRC3). This is done with a create macro in assembly language or a create function in C language, and CRC2 uses a form of the function that requests an immediate activation of the new entry.

Assembly language example:

```
LA    R14,48        Number of bytes to pass to the new entry

LA    R15,EBW020    Location of data to pass

CREMC CRC3          Program to be activated to process the new entry
```

C language example:

```
cremc(sizeof(packedacct), cremargs, CRC3);
                                /* Create entry CRC3 */
```

CRC3 then continues with the mainline process.

Step 18 CRC2 processing is now complete. In assembler, it issues a BACKC macro, which returns control to CRC1. In C language, control is returned by a return statement. Note that it is not necessary to return the main storage blocks on levels 4 and 5 because TPF does this automatically unless instructed not to do so.

All that remains is to send a message to the terminal operator indicating that the credit may be granted.

The first requirement for an assembly language program to send a message is to build the message block. The first or only segment of the message must be in main storage, attached to any level of the ECB. CRC1 requests a 127-byte block from TPF on level 6. The same data macro (AM0SG) is used for output and input messages. The data macro is issued, and register 3 loaded with the pointer from level 6.

```
GETCC D6,L0
AM0SG REG=R3
L     R3,CE1CR6
```

The header of the record is initialized with the record ID (‘OM’), the forward chain field is zeroed, and the character count and text are entered.

In a C language program, CRC1 uses the `puts` function to output the message to terminal.

Step 19 CRC1 builds the output RCPL for assembly language.

The `puts` function is not ANSI or ISO compliant. Its action is defined by site installation. The input RCPL, received from the COMM SOURCE program at EBW000, has not been disturbed and contains the necessary data for output. The first two fullwords containing destination and origin are exchanged; destination becomes the LNIATA and CPU ID of the terminal, origin becomes CRED. The control field bits are modified as described in the DSECT RC0PL for assembly language or the `c$rc0pl.h` header for C language.

Assembly language program CRC1 loads register 3 with the address of the RCPL (EBW000) and requests TPF to send the message by issuing the ROUTC macro.

Example:

```
ROUTC LEV=D6,LIST=R3
```

This specifies that the message is in core at level 6 and register 3 points to the RCPL.

Step 20 Processing for the entry is now complete; the application has no further function to perform. CRC1 stops by calling the EXITC macro in assembly language or the `exit` function in C language. TPF removes the entry from the system and releases the main storage block containing the ECB. If any main storage blocks are attached to the ECB, they are released at exit.

The preceding process by no means presents a definitive review of all application facilities. It is meant to be illustrative and to give a practical framework into which to fit the detailed descriptions that follow.

TPF Data Structures and System Services

This chapter describes some commonly used data structures and system services in TPF application programs. Familiarity with this material will enhance your understanding of the events that occur when TPF processes an entry. These concepts form the basis for the discussion in “TPF Application Program Interface Functions” on page 227 of the application programming interface functions for the C language support.

See “Language Structures and Case Guidelines” on page 1 for an understanding of why we provide both assembly language and C language structures in some cases and only C language structures in others.

Entry Control Block

The entry control block (ECB) is the primary interface between TPF and the application program. TPF creates an ECB for every message entering the system and follows the processing of that message throughout its life in the system. Additional entries (ECBs) may be created at the request of the application to subdivide the processing. Among other things, the ECB contains an activation number, switches, counters, two limited work areas, and pointers to additional data in main storage or in file storage. The ECB thus enables the application to control its processing and to request services from TPF to control file storage and main storage, transmit messages, and pass control among program segments.

The assembler application accesses its ECB through general register 9. This register is reserved in the TPF system for this function. At all times there is a pointer in register 9 to a unique ECB that is being referenced by the application program in control. The program, in turn, is pointed to by another dedicated register, register 8.

The C or C++ language application accesses its ECB through linkage generated by the IBM C and C++ compiler products on the System/390 platform. The `ecbptr` macro is available to the application programmer to access information in the ECB. The `ecbptr` macro returns a pointer to a structure of type `eb0eb` to the currently executing ECB. This structure is defined in the `c$eb0eb.h` header.

The ECB enables application programs to be reentrant, a crucial feature of TPF’s high performance. To be reentrant, an application program refers to switches, counters, and pointers in the ECB and not in the program itself.

TPF dynamically maintains all references to system storage, register content and usage in the ECB. Consequently, one main storage copy of a program, by referring to different ECBs, is able to process many different messages in various stages of progress.

Whether in assembly language or C language the ECB is defined in great detail; almost every byte has its own symbolic name. The assembly language DSECT that defines the symbolic names and displacements in the ECB is named EB0EB. The `c$eb0eb.h` header defines the symbolic names and displacements in the ECB for C language.

In assembly language the EB0EB macro does not have to be coded explicitly because it is called by the BEGIN macro, which must be the first statement in every

ECB-controlled assembler language program. In C language, if a program refers to ECB fields, you must include the `c$eb0eb.h` header, either explicitly or implicitly.

In C language the last 3 or 4 characters of the symbolic names specified in `c$eb0eb.h` are used as dump tags to identify ECB fields in a main storage dump. For example, W032 in a main storage dump identifies `ebw032` in application work area 1.

Note: The corresponding ECB fields are listed in lowercase in the C language and uppercase in assembler.

There are 3 categories of ECB fields:

- Fields assigned as a work area for application programs
- Fields used by TPF system programs and application programs for passing main storage and data base references
- Fields used exclusively by TPF as save areas and control fields for monitoring the entries.

Table 1 summarizes the fields of greatest importance in application programming. Study the EB0EB DSECT in assembly language or the `c$eb0eb.h` header in C language for an understanding of the ECB fields in this critical block.

Table 1. Summary of the Entry Control Block Application and Interface Areas

Field(s)	**Area**

`ebw000-ebw103`
`ebsw01-ebsw03`
`ebrs01`
`ebcm01-ebcm03`
`eber01`
	Application interprogram work area 1

`ce1fa0-ce1faf`	File address reference words (FARW)
`ce1cr0-ce1crf`	Core block reference words (CBRW)
`ce1fx0-ce1fxf`	File reference address word extensions (FAXW)
`ce1sud`, `ce1sug`	System error indicators
`ce1rda-ce1svp`	TPF register save area
`ce1ars`	User (application) register save area
`ebx000-ebx103` `ebxsw1-ebxsw7`	Application interprogram work area 2
`ce1usa`	User area

Linking ECBs and Their Common Services

The TPF system uses the first 8 bytes of the ECB to link together ECBs requesting the same service, and for the branch address of the system processing routine when the ECB is activated.

Work Areas

Application and design requirements determine the use and format of these areas. You may use the work areas for switches, indicators, and other temporary storage needs unique to each processing program segment, or for passing information from program to program. You can make your application programming easier if each application program in the processing chain specifies in its documentation how each work area will be used.

The work areas must be shared by all programs processing the entry. Two important design considerations for creating a TPF application are to assure a workable protocol and an optimum use of the space for data that must be saved or passed along.

You can use the following ECB work areas for application programming:
- Application work areas
- User work area
- User application register save area.

Application Work Areas

There are two application work areas, each 112 bytes long, consisting of a 104-byte scratch area followed by an 8-byte bit-switch area.

The first work area begins at `ebw000`. Each byte of its scratch area is named sequentially beginning with `ebw000` and ending with `ebw103`. The second work area begins at `ebx000`. Each byte of its scratch area is named sequentially beginning with `ebx000` and ending with `ebx103`.

The last 8 bytes in each of these work areas are intended for use as program switches. The program switch names and their standard usage conventions are as follows:

First work area:

Name	**Usage Convention**
`ebsw01—ebsw03`	Interprogram switch to specify various conditions among programs
`ebrs01`	Used to pass error information among program segments
`ebcm01—ebcm03`	Intraprogram switch to specify various conditions within programs
`eber01`	Used to pass error information within program segments.

Second work area:

Name	**Usage Convention**
`ebxsw1—ebxsw7`	No convention.

These conventions are recommended. However, applications not requiring this logic are free to use these bytes in any way desired.

OPZERO initializes the switches—that is, the last eight elements—in both work areas, to zero when it creates the ECB. The first 104 elements of the work areas are not initialized, so their contents are unpredictable. (COMM SOURCE will later place the routing control parameter list, occupying at least 12 bytes, at the beginning of the interprogram work area. See “Routing Control Parameter List” on page 38 for further information.)

User Work Area

The user work area is 2752 bytes long and begins at `ce1usa` as listed in Table 1 on page 26.

User Application Register Save Area

The user application register save area is for saving registers in assembler application programming. It begins at `ce1ars` and is 10 fullwords long (registers 14 and 15 and registers 0 through 7).

Data Levels

Each entry has the capability of referencing 16 data blocks concurrently in the ECB. The ECB maintains these references in sixteen data levels numbered hexadecimally from 0 to F (often referred to as D0–DF). As used by the TPF system, the term *data level* does not imply any priority assigned to the level number or any nesting. A data level is simply a series of doubleword reference/control fields for data blocks that can be used however the application requirements dictate.

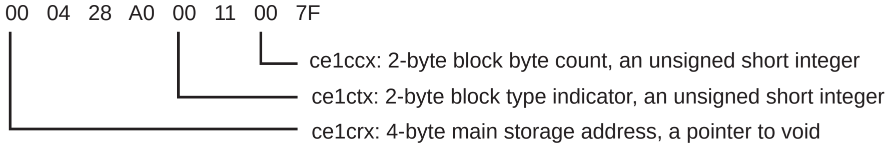
There are three sets of doubleword references for ***each*** data level. They are, in order of their physical position in the ECB:

- File address reference words (FARW)
- Core block reference words (CBRW)
- File address extension words (FAXW).

The FARWs and CBRWs are of primary importance to your application.

Core Block Reference Words

There is an 8-byte core block reference word (CBRW) for each data level, each of which the C language treats as a pointer followed by two unsigned short integers. The CBRW is used to store the main storage address and control information about main storage blocks used by the entry. TPF system programs format the CBRW whenever a main storage block is attached to or detached from the ECB. The format consists of the main storage address (4 bytes), the block type indicator (2 bytes), and the block byte count (2 bytes). The main storage block address has the same label as the CBRW. The block type indicator and the block byte count have their own labels. In the following example, `x` is the data level number, a hexadecimal digit from 0 to F.


00 04 28 A0 00 11 00 7F
ce1ccx: 2-byte block byte count, an unsigned short integer
ce1ctx: 2-byte block type indicator, an unsigned short integer
ce1crx: 4-byte main storage address, a pointer to void

The block type indicator (`ce1ctx`) specifies the size of the block attached to the ECB at level `x`.

0001	No block attached	
0011	127-byte block attached	(L0)
0021	381-byte block attached	(L1)
0031	1055-byte block attached	(L2)
0051	4095-byte block attached	(L4)

The block byte count (`ce1ccx`) specifies the number of bytes in the block that the program can access.

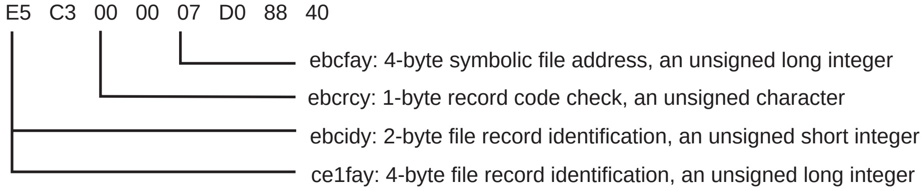
It is important to note that only the block type indicator accurately reflects the status of the level at any given time. If `ce1ctx` contains X'0001', then the data in the rest of the CBRW for level `x` is not meaningful. TPF does not initialize `ce1crx` and `ce1ccx` after a block is released. Only if `ce1ctx` contains X'0011', X'0021', X'0031' or X'0051' is a currently valid main storage address in the level `x` CBRW.

TPF updates the CBRW as a result of an application request to obtain or release a main storage block or to read or write a file record. Application programs must never modify the CBRW, but may refer to it to determine data level status and the address of the acquired block.

File Address Reference Words

There is an 8-byte file address reference word (FARW) for each ECB data level, each of which the C language treats as an unsigned long integer followed by four unsigned characters. The FARW is used to record the file address and control data related to file I/O for each data level. There may be a main storage block on a given level without any file activity, in which case the FARW will not be used. When there is file activity the FARW will be used for the record identification (2 bytes), record code check (1 byte), and symbolic file address (4 bytes); 1 byte is unused. In the following example, *y* is the data level number, a hexadecimal digit from 0 to F.

Example:


E5 C3 00 00 07 D0 88 40
ebcfay: 4-byte symbolic file address, an unsigned long integer
ebcrcy: 1-byte record code check, an unsigned character
ebcidy: 2-byte file record identification, an unsigned short integer
ce1fay: 4-byte file record identification, an unsigned long integer

When the application program requests TPF, via a macro or a library function call, to read or write a record, the FARW at the specified data level must be set up as described in “File Storage Access” on page 257.

File Address Extension Words

The file address extension word (FAXW) is used to pass information between TPF online systems and MVS, using DASD files called *general data sets*. See *TPF Database Reference* for more information about the format and use of the FAXW.

Data Event Control Blocks

You can use data event control blocks (DECBs) as an alternative to using standard ECB data level information. ECB data level information is used to specify information about I/O request CBRW, FARW, and FAXW fields. Although a DECB does not physically reside in an ECB, the DECB fields specify the same information without requiring the use of a data level in the ECB. All the same requirements and conditions that apply to the CBRW, FARW, and FAXW fields in the ECB also pertain to the same field information in the DECB.

Figure 4 on page 30 shows the DECB application area and the fields inside the DECB:

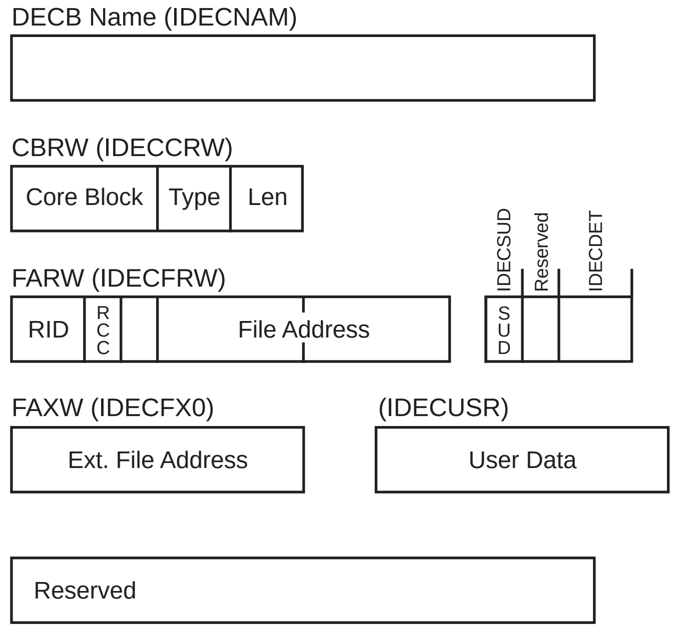


Figure 4. DECB Application Area

DECB Fields

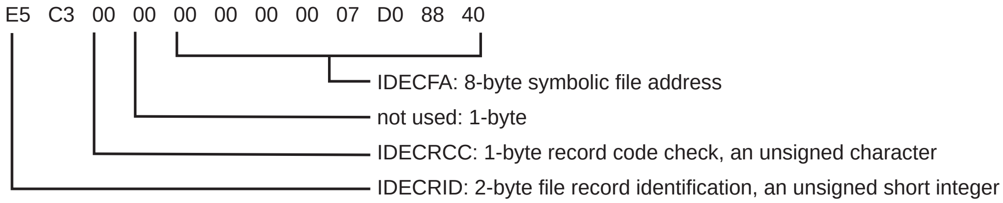
The DECB fields are used as follows:

IDECNAM	Contains the name of the DECB.
IDECCRW	Corresponds to an ECB core block level. IDECDAD (Core block) Contains the address of a core block. IDECCT0 (Type) Contains the core block type indicator or X'0001' to signify there is no core block attached. IDECDLH (Len) Contains the data length.
IDECFRW	Corresponds to an ECB FARW. IDECRID (RID) Contains the record ID for a FIND/FILE request. IDECRCC (RCC) Contains the record code check (RCC) value for a FIND/FILE request. IDECFA (File Address) Contains the file address for a FIND/FILE request.
IDECSUD	When the FIND/FILE request is completed, this field will be set to the SUD error value or zero if there is no error.
IDECDET	Contains the number of core blocks currently detached from this DECB.
IDECFX0	Contains the FAXW information.
IDECUSR	Contains the user data.

8-Byte File Address Support

Although an ECB data level and a DECB are alike, there is a difference in the FARW. The IDECFA field, which contains the file address, has been expanded to 8 bytes in the DECB. This expansion allows 8-byte file addressing in either 4x4 format or FARF6 mode. 4x4 format provides for standard 4-byte file addresses (FARF3, FARF4, and FARF5) to be stored in an 8-byte field. FARF6 is the exploitation of 7 of the 8 bytes in the file address.

A 4-byte file address in 4x4 format resides in the low-order 4 bytes of the IDECFA field. The high-order 4 bytes of the IDECFA field contain an indicator (a fullword of zeros) that classifies it as a valid 4x4 format address. The high-order 4 bytes of a FARF6 file address is a nonzero value. When there is file activity, the FARW will be used for the record identification (2 bytes), record code check (1 byte), and a symbolic file address (8 bytes); 1 byte is unused. For example:


E5 C3 00 00 00 00 00 00 07 D0 88 40
IDECFA: 8-byte symbolic file address
not used: 1-byte
IDECRCC: 1-byte record code check, an unsigned character
IDECRID: 2-byte file record identification, an unsigned short integer

Referencing Data Blocks

As previously discussed in “Data Levels” on page 28, data levels can reference 16 data blocks concurrently in the ECB. If you choose to use DECBs, you are not restricted to 16 data blocks. DECBs can be acquired dynamically by a single ECB by using the `tpf_decb_create` function or the DECBC macro. The storage, which will hold the DECB, comes from the 1-MB private area of the ECB. Therefore, the number of DECBs that the ECB is restricted to is limited only by the amount of storage in the private area that is dedicated to the DECB. See the *TPF C/C++ Language Support User’s Guide* for more information about the `tpf_decb_create` function and *TPF General Macros* for more information about the DECBC macro.

Using Symbolic Names

Using DECBs allows you to associate symbolic names with each DECB. This allows different components of a program to easily pass information in core blocks attached to a DECB. Each component only needs to know the name of the DECB where the information is to be found to access it. However, functions that support the use of a DECB (such as `file_record_ext`, `find_record_ext`, and so on) will only accept a DECB address as a valid reference to a DECB. If an application does not maintain the address of a particular DECB and, instead, maintains the name of the DECB, the caller will first have to issue the `tpf_decb_locate` function to obtain the address of the DECB. The resulting DECB address can then be passed on the subsequent function call.

Accessing File Records with a DECB

All types of applications can use DECBs. Application programming interfaces (APIs) have been added to allow TPF programs to access file records with a DECB instead of an ECB data level. However, only a subset of the existing macros and C functions that currently reference ECB data levels accept a DECB in place of an ECB data level. Macros and C functions that use a file address will first verify that it is a valid address in 4x4 format or FARF6 mode. If the address is not valid, a system error will occur. See *TPF General Macros* and *TPF System Macros* for more information about general and system macros that were added or changed for TPF DECB support.

Applications that call the following functions (using 8-byte file addresses or DECBs in place of ECB data levels) must be compiled with the C++ compiler and they must create linkage to the service routines in the CTAD dynamic link library (DLL):

- `attac_ext`
- `creec`, `__CREEC`
- `cretc_level`, `__CRETCL`
- `crusa`
- `csonc`
- `detac_ext`
- `file_record_ext`
- `find_record_ext`
- `gdsnc`
- `gdsrc`
- `getcc`
- `getfc`
- `levtest`
- `rcunc`
- `relcc`
- `relfc`
- `rlcha`
- `sonic`
- `swisc_create`
- `tpf_decb_create`
- `tpf_decb_locate`
- `tpf_decb_release`
- `tpf_decb_swapblk`
- `tpf_decb_validate`
- `tpf_esfac`
- `tpf_fac8c`
- `tpf_faczc`
- `tpf_fa4x4c`
- `tpf_rcrfc`
- `unfrc_ext`.

New or existing applications that only use ECB data levels can still use the C compiler to create linkage to these TPF functions in the CTAL DLL. See the *TPF C/C++ Language Support User's Guide* for more information about these functions.

Error Handling

With TPF DECB support, each DECB has a detailed error indicator byte (IDECSUD). The existing `ce1sug` byte will include any errors that occur on a DECB-related I/O operation.

Functional Flow

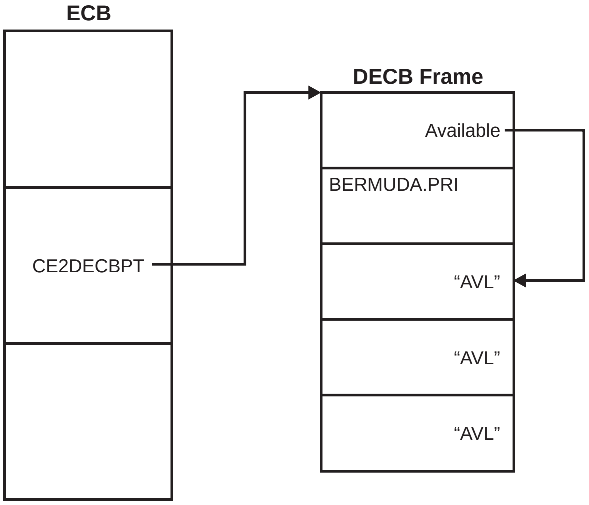
The flow of a DECB through the TPF 4.1 system can be very different depending on how it is used by the application. The DECB acts as an interface between the application and the TPF control program, and contains relevant information about core block and file I/O requests. The following example shows how TPF DECB support works by providing a sample application and describing how DECBs are referenced by the TPF 4.1 system.

Assume there is a TPF application called Bermuda. When an airline passenger purchases a one-way ticket to the island of Bermuda, this application is activated to update a database that maintains a list of every one-way passenger. The Bermuda application consists of two program segments, SUNN and SAND. The SUNN segment will validate the database update request and verify that the passenger record does not already exist in the current passenger list. Once the request has been validated, SUNN will forward the request to the SAND segment, which will add the name of the new passenger (A. Traveller) to the existing passenger list.

1. SUNN is entered by a reservation application with a copy of the passenger record of A. Traveller attached on data level 0 (D0) of the ECB. After verifying that there is a passenger record attached to D0, SUNN allocates a DECB, which will be used to hold the primary database record for Bermuda. The following example shows the DECB allocation:

   ```
   Assembler:   IDECB REG=R1
                DECBC FUNC=CREATE,DECB=(R1),NAME=DECBPRIME
                ...
                DECBPRIME DC CL16'BERMUDA.PRI'

   C++:         TPF_DECB *decb;
                DECBC_RC rc;
                ...
                decb = tpf_decb_create ("BERMUDA.PRI", &rc;);
   ```

   Because there were no DECBs previously associated with this entry, a 4-K frame will be obtained from the ECB private area, which becomes a DECB frame. Multiple DECBs will be carved from a single DECB frame. The address of the first DECB frame will be stored in page 2 of the ECB in the CE2DECBPT field. It is also possible to have a single ECB with more than one DECB frame. Each successive DECB frame will be forward-chained from the previous frame.



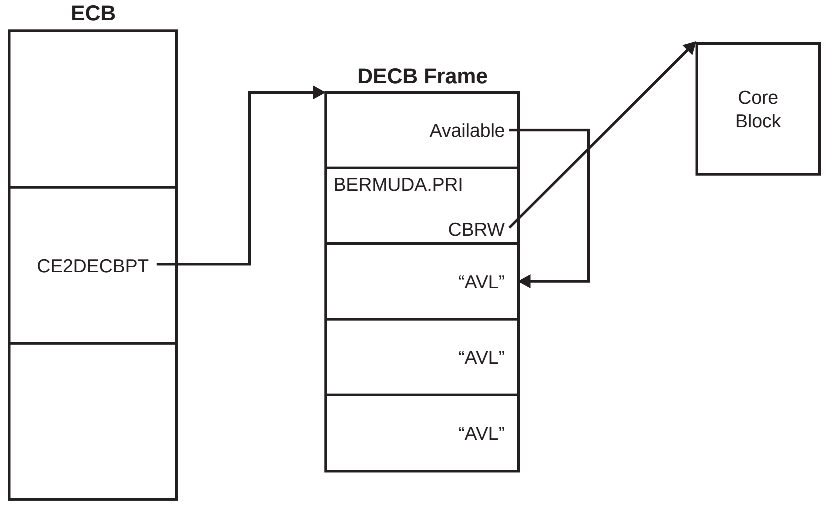
2. Now that a DECB is available for use, SUNN will attempt to calculate the file address of the primary record for the Bermuda database. The following example shows a call to the file address compute program (FACE) to determine the correct file address:

```
Assembler: IFAC8 REG=R7
           LA R7,EBX000
           MVC IFACORD,=XL8'150'
           MVC IFACREC,=CL8'#BERMUDA'
           MVI IFACTYP,IFACFCS
           FAC8C PARMS=(R7)

C++:      TPF_FAC8 *fac8_parms;
          ...
          fac8_parms = (TPF_FAC8 *)&ecbptr()->ebx000;
          fac8_parms->ifacord = 0x150;
          memcpy (fac8_parms->ifacrec, "#BERMUDA", 8);
          fac8_parms->ifactyp = IFAC8FCS
          tpf_fac8c (fac8_parms);
```

3. SUNN now has the file address for the primary record and will issue a FIND request to bring the record into working storage and obtain an exclusive lock of the record for this ECB. The following example shows how the FIND-type call might look:

```
Assembler: MVC IDECFA,IFACADR
           MVC IDECRID,=CL2'AB'
           XC  IDECRCC,IDECRCC
           FIWHC ,DECB=(R1),ERROR_BRANCH

C++:       decb->fa = fac8_parms->ifacadr;
           find_record_ext (decb, NULL, "AB", '\0'
                            HOLD_WAIT, FIND_DEFEXT);
```

4. When the FIND request ends either successfully or unsuccessfully, the DECB will be updated accordingly. For a successful call, the SUD value in the DECB will be cleared and a core block will be attached at the CBRW of the DECB.

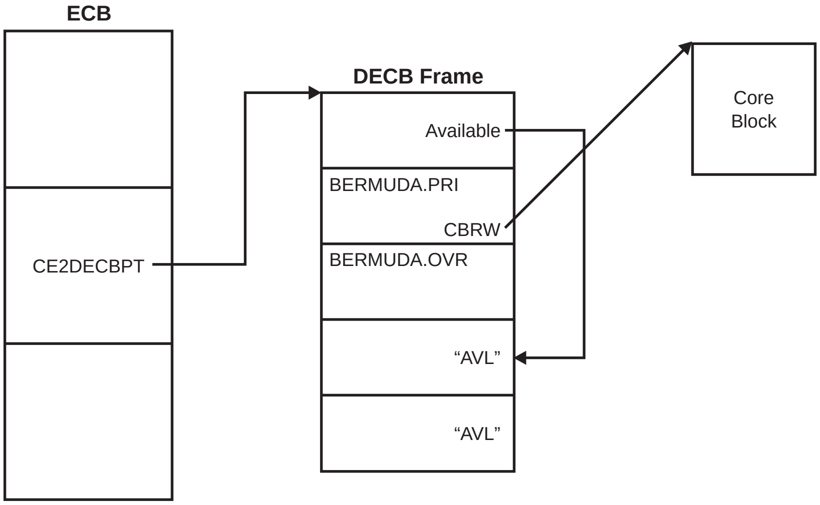


5. Now that the primary record has been read in from the database, SUNN enters segment SAND to complete the processing. SAND will search the primary record for an available entry slot to which the passenger record of A. Traveller can be added. If there are no available entries in the primary record, an overflow record will be obtained. To obtain the overflow record, a new DECB must be created. The following example shows the allocation of a new DECB:

```
Assembler:  DECBC FUNC=CREATE,DECB=(R2),NAME=DECBOFLW
            ...
            DECBPRIME DC CL16'BERMUDA.PRI'
            DECBOFLW DC CL16'BERMUDA.OVR'

C++:        TPF_DECB *prime, *overflow;
```

```
DECBC_RC rc;
...
overflow = tpf_decb_create ("BERMUDA.OVR", &rc);
```

The first available DECB in the DECB frame is dispensed to the application. There are now two DECBs in the single DECB frame, which are marked as in use by the ECB.

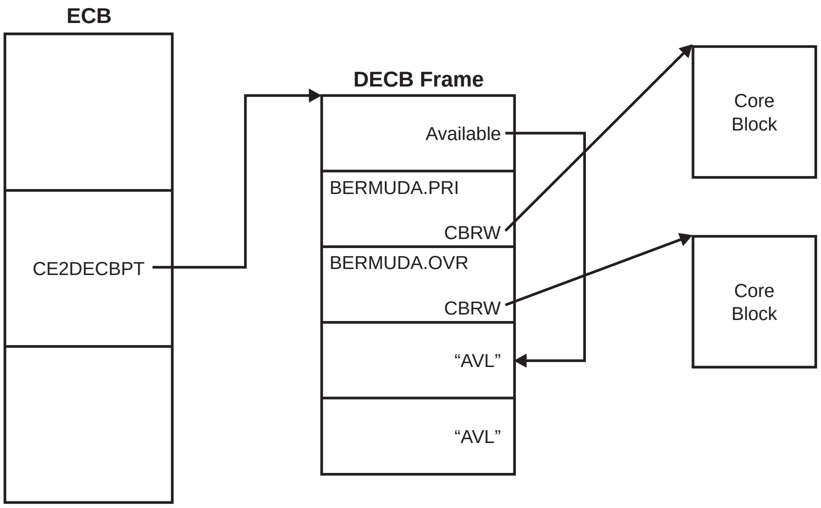


6. SAND must obtain a file pool record, which will serve as the new overflow record in the Bermuda database. SAND must then locate the DECB created by SUNN, which contains the primary record so that the pool record can be chained to it. The following example shows how the file pool record is obtained and how the DECB containing the primary record is located:

```
 Assembler:  GETFC ,DECB=(R2),ID=CL2'AB',BLOCK=NO
             DECBC FUNC=LOCATE,DECB=(R3),NAME=DECBPRIME
             ...
             DECBPRIME DC CL16'BERMUDA.PRI'
             DECBOFLW DC CL16'BERMUDA.OVR'

 C++:        TPF_FA8 pool_addr;
             ...
             pool_addr = getfc (overflow, GETFC_TYPE0, "AB",
                                GETFC_NOBLOCK, GETFC_SERRC);
             prime = tpf_decb_locate ("BERMUDA.PRI", &rc);
```

7. To build the overflow record, SAND must get a core block and attach it to the DECB where the overflow pool file address was obtained. The following example shows a call to the get core routine:

```
Assembler:  GETCC ,DECB=(R2),L4,FILL=00

C++:        getcc (overflow,
                  (enum t_getfmt)(GETCC_TYPE+GETCC_FILL),
                   L4, 0x00);
```

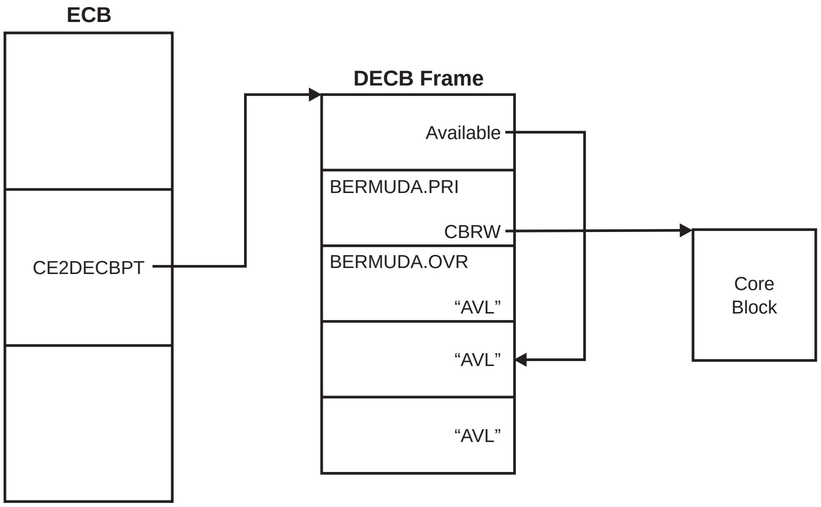


8. SAND must now copy the relevant information from the passenger record of A. Traveller, which is attached on ECB data level D0 to an available entry in the overflow record attached to the DECB. After copying the information, a FILE macro request must be issued to update the Bermuda database with the information of the new passenger. The following example shows the FILE request:

```
Assembler:  FILEC DECB=(R2)

C++:        file_record_ext (overflow, NULL, "AB", '\0'
                             NOHOLD, FILE_DEFEXT);
```

9. Now that the overflow record is no longer being referenced by the ECB, SAND may choose to release the DECB that was allocated to hold the record. The following example shows the call to release the DECB, which once held the new overflow record of the Bermuda database:

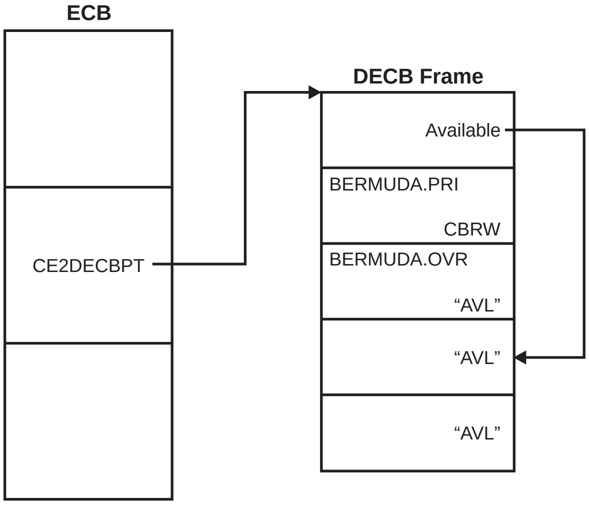
```
Assembler:  DECBC FUNC=RELEASE,DECB=(R2)

C++:        tpf_decb_release (overflow)
```



10. Having filed down the updated overflow record, the SAND segment can now update the primary record with the file address of the overflow record to chain them together. After performing this update, SAND can now issue a FILE

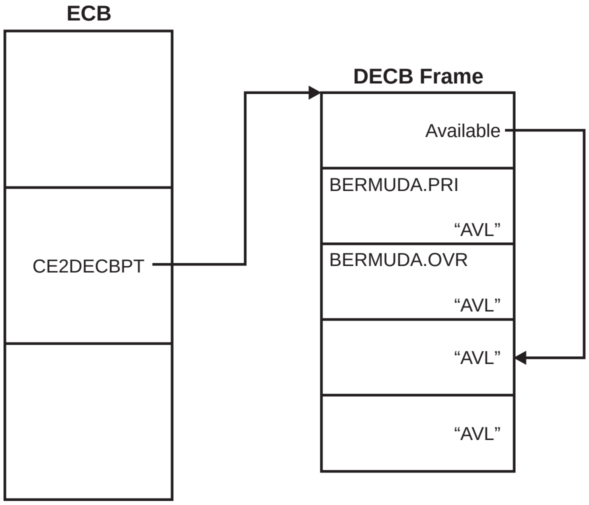
request to update the primary record in the database and release the lock, which SUNN had previously obtained on the record. The following example shows this FILE request:

```
Assembler:  FILUC DECB=(R3)

C++:        file_record_ext (prime, NULL, "AB", '\0',
                             UNHOLD, FILE_DEFEXT);
```



11. After performing its final operation on the Bermuda database, SAND may choose to release the final DECB, which was being used before exiting the ECB. Even if the final DECB is released by the application, the DECB frame will remain attached to the ECB in preparation for when the next DECB will be created. The DECB frame will be released when the ECB is finally exited.

```
Assembler:  DECBC FUNC=RELEASE,NAME=DECBPRIME
            ...
            DECBPRIME DC CL16'BERMUDA.PRI'
            DECBOFLW DC CL16'BERMUDA.OVR'

C++:        tpf_decb_release ("BERMUDA.PRI");
```



I/O-Associated Unusual Conditions

Two fields in the ECB are used to indicate unusual conditions associated with I/O requests:

- `ce1sug` is a 1-byte gross or summary indicator of all unusual conditions occurring on any level.
- `ce1sud` is a series of 16 indicators—1 byte for each level.

The application should never clear or modify `ce1sug`, and should only modify `ce1sud` immediately after an I/O request has ended. TPF resets `ce1sug` and `ce1sud` after each `waitc` function call.

TPF General Macros discusses these indicators in detail.

Note: See "Error Handling" on page 32 for information about unusual conditions for DECBs.

Areas Used by the TPF system

Some areas of the ECB are used exclusively by the TPF system, but can be useful in error analysis. More about error analysis can be found in *TPF Program Development Support Reference* as well as "TPF Testing Environment for Assembly Language" on page 283 in this publication.

TPF Register Save Area

The primary function of the TPF register save area is to save registers 14, 15, and 0 through 7 (the general application use registers) plus register 8 (the application program base register) whenever the application program surrenders control by issuing a control program macro. When control is returned to the application program, the registers are restored from this area.

A main storage dump shows the contents of the registers at the time of the last control program macro, which can be helpful in pinpointing the section of code that was processing at that time.

Control Program Save Area

TPF saves the program status word whenever the operational program surrenders control as a result of a wait type function, and restores it from this area when that program regains control. This area contains pertinent information associated with the application state.

User Register Save Area

The user register save area, which is also known as the application register save area, generally is used in assembler application programming to save register contents when they are being passed between programs. The user register save area is distinct from the TPF register save area.

Routing Control Parameter List

A routing control parameter list (RCPL) is associated with each TPF input or output message. The RCPL provides information about the origin, destination, and characteristics of the message. TPF programs use this information to determine the routing required for the message. Application programs, for input messages, use the RCPL to determine the message characteristics and its origin. The application's input message editor must look at the RCPL to determine if the input is a normal new message, a resubmitted input message resulting from failure of a reply to the original, or a returned output message resulting from a failure to transmit successfully to its destination.

For output messages, the application program generates the RCPL to provide the TPF programs with the information necessary to route the message to its destination.

The RCPL is passed to the application starting at `ebw000` of the ECB work area. The length of the RCPL is either the 12-byte basic format or 16-byte extended format. The extended format provides for an additional optional general data area of up to 82 bytes. The application must specify at implementation time which format will be used. This is done with the RCPL parameter of the MSGRTA macro in SIP. (See *TPF System Generation*.)

RCPL Data Area

The header that defines the RCPL is DSECT RC0PL or header `c$rc0pl.h`. The data area of the RCPL contains three major fields, whether input or output, base or expanded format:

Field	**Size**
Destination	4 bytes
Origin	4 bytes
Control	4 or 8 bytes, depending on the RCPL format

RCPL Destination Field

This field specifies the destination address of the message. Its contents depend on the type of destination, that is, whether it is an application program or a terminal. If the destination is an application program, then this field contains the 4-character EBCDIC application name. If the destination is a terminal, then this field will contain the TPF terminal address, but will be unique for SNA and non-SNA systems. When the destination is an SNA logical unit this field will contain:

- An identifier of the logical unit to which the message is being sent. This 3-byte subfield is called the network addressable unit (NAU), the resource identification (RID) of the logical unit or node, or the session control block identifier (SCB ID).
- The 1-character identification of the CPU to which the logical unit is directly connected.

When the destination is a non-SNA terminal, this field will contain:

- The 3-byte line, interchange, and terminal address (LNIATA).
- The 1-byte identification of the CPU to which the terminal is directly connected.

RCPL Origin Field

This field specifies the origin address of a message. As with the destination field, the contents of this field can be either an application name (for application to terminal or application to application messages) or a TPF terminal address (for terminal to application messages) in the forms described in the destination field above.

RCPL Control Field

This field contains information describing the message origin and destination along with various control indicators. The contents of the control bytes within this field vary depending on whether this RCPL is being passed to the TPF router package (via the ROUTC macro or the `routc` function) by an application program, and also on whether base or expanded format is used.

In the basic format the control field consists of 4 bytes, the first 3 bytes of which are of primary significance to the application:

- Byte 0 is summarized in Table 2.
- Byte 1 is used for messages to or from a terminal and contains the terminal type character from the TRMEQ assembler macro (see *TPF General Macros*), except for 3600/4700 devices, in which case the byte contains X'00'.
- Byte 2 also is summarized in Table 2.
- Byte 3 is used exclusively by the TPF system.

Note: As shown in the following table, bit meanings in control bytes 0 and 2 vary in some cases for input and output. The table should be used with `c$rc0pl.h`, because the programming considerations cannot all be precisely defined in the table. You can find additional information in *TPF ACF/SNA Data Communications Reference* and in *TPF Data Communications Services Reference*.

In the expanded RCPL, there are four additional control bytes:

- Byte 4 contains 2 specialized indicators related to use of the expanded RCPL for message recovery and 3600/4700 batch LU support.
- Byte 5 is reserved.
- Bytes 6 and 7 are used, optionally, to define an attached general data area of up to 82 bytes. The use of the general data area is beyond the scope of this publication. See the RC0PL DSECT or the `c$rc0pl.h` header for more information.

Table 2. Application Use of RCPL Control Bytes 0 and 2

Byte	Bit	Use	Set	Meaning On Input	Meaning On Output
0	0	Destination type	0 1	Destination is terminal Destination is application	Same as input Same as input
	1	Origin type	0 1	Origin is terminal Origin is application	Same as input Same as input
	2	Message type	0 1	Not used Not used	Reply to input message Unsolicited message
	3	Message priority	0 1	TPF use only TPF use only	TPF use only TPF use only
	4	Message recovery	0 1	Normal input message Returned output message	Return to application if undeliverable Queue message if undeliverable
	5	SLC usage	0 1	Not used Not used	Output less than 4000 bytes Output exceeds 4000 bytes
	6	Terminal address format	0 1	LNIATA address RID address	Same as input Same as input
	7	RCPL format	0 1	Basic 12 bytes Expanded format	Same as input Same as input
2	0	Release output	0 1	Not used Not used	Release output pool file record Don’t release output pool file record
	1	UIO ROUTC	0 1	Not used Not used	ROUTC not issued by UIO ROUTC issued by UIO
	2	Resubmitted input	0 1	Not used Not used	TPF use only TPF use only
	3	Message format	0 1	Not used Not used	AMSG format OMSG format

Table 2. Application Use of RCPL Control Bytes 0 and 2 (continued)

Byte	Bit	Use	Set	Meaning On Input	Meaning On Output
	4	Brackets	0 1	No brackets (SNA) Begin (CONTINUE) brackets	Bracket state unchanged End bracket state
	5	Release input file	0 1	Message not recoverable Message recoverable	Don't release input file record Release input file record
	6	Change direction	0 1	Normal input Resubmitted (possible duplicate msg.)	No change in direction of data flow Change in direction of data flow
	7	Output FM header	0 1	No FM header FM header in text	No FM header FM header in output text

RC0PL Data Macro

The data macro name for the DSECT that defines the RCPL is RC0PL. This DSECT contains labels to refer to each of the fields as well as labels equated to the necessary values for testing and resetting each of the bit indicators. Since new applications have the option of using the base or expanded format, the macro has a keyword for specifying which format the program requires.

Example:

```
RC0PL REG=Rxx|ORG=addr (,FORM=BASE/XPND)
```

Where:

- RC0PL is the data macro name
- REG= is the keyword used to specify a symbolic register name (Rxx) valid for assignment to the USING statement as the base register for DSECT RC0PL.
- ORG= can be used instead of REG= when the RCPL will be defined as a relocated value (for example, ORG=EBW000).
- FORM= is the keyword used to specify the basic or expanded format of this macro. Valid operands are:
  – BASE for the basic 12-byte format.
  – XPND for the expanded format which is a minimum length of 16 bytes.

  BASE is the default value.

Activating the Application

This section discusses the log processor and conditions at activation.

Log Processor

When a terminal operator enters a message intended, for example, for a credit verification application, it is not practical for TPF to analyze the text of the message to determine its intended destination. Large TPF systems may host thousands of terminals and dozens of applications, each of which might have scores of unique message types. The TPF log processor resolves this problem and allows any terminal to be connected to any application in the network at a given time. This is not a hardware connection; by the use of the system tables and programs all data entered at the terminal for the duration of the connection is delivered to the application requested.

The connection is established and terminated by log messages from the originating terminal. A terminal operator gets access to an application with a login message and terminates the connection with a logout message.

The system tables that are used to maintain terminal to application logging are mostly transparent to the application, so we will discuss them only briefly.

Application Name Table (ANT)

This table contains an item for every application in the system to which a terminal may log. Each item has the 4-character application name that would be used in the login message and status information about that application. Each item contains a pointer to that application's entry in the routing control application table (RCAT).

Routing Control Application Table (RCAT)

The RCAT has more detailed information about each application and pointers that enable efficient routing of messages. Included in each item are:

- The status of the application
- Whether it uses the base or expanded RCPL
- The address of the application input message editor, which is the program segment to be activated to begin processing of input messages.

Operator Security: The RCAT also indicates that the sign-in or sign-out function is supported by the application.

This level of security is designed uniquely by the application. This security may be needed for accounting purposes, for applications performing multiple functions each of which require unique security codes or for a variety of reasons known only to the application.

Although sign-in or sign-out is not a TPF operation, the log processor includes an interface for it via a sign-in or sign-out procedure. If the RCAT entry for the application indicates sign-in/sign-out is required, this interface allows the application to notify the log processor of successful sign-in. Once the log processor is so notified, it no longer monitors messages from that terminal. They are passed directly to the application, until such time as the operator signs out and the log processor is notified.

The interface to the log processor is in the form of an application to application data transmission where:

- The origin of the data is the application itself.
- The destination of the data is application CLG*x* (*x* is the ID of the CPU in direct control of the terminal).
- The text of the data is the address (LNIATA/CPUID) of the originating terminal.

Resource Vector Table (RVT)

The RVT is used for SNA terminal definition and control. There is an item in the record for each network resource in the network. When an SNA terminal logs in to an application, the RVT item is updated with an index value that serves two purposes. It indicates that the terminal is logged to an application, and it points indirectly to the RCAT item that gives the routing data for that application. When the application being logged on to is a non-SNA application, the WGTA table will also be updated.

WGTA Table (WGTA)

This table serves a purpose for non-SNA terminals similar to that of the resource vector table for SNA. When a terminal logs in to an application, the WGTA item for that terminal is updated with the index value pointing to the routing control information. WGTA contains either the RCB or AAA addresses. (See "Routing Control Block" on page 48 and "Agent Assembly Area" on page 49).

Conditions at Activation

To recap, once a terminal is logged to an application, subsequent messages from that terminal are routed directly to the application. The application message editor for the application is pointed to by the RCAT. When COMM SOURCE completes its processing of the input message, it passes control to that program. Figure 1 on page 13 shows the conditions at that point.

In summary:

- The ECB is in main storage and register 9 is loaded with its main storage address.
- The input message (or the first block of the message) is in main storage in AMSG format and is pointed to by data level 0 of the ECB.
- The RCPL will be in the ECB starting at location `ebw000`.
- For messages arriving from terminals (or intended for RCB-dependent applications), the RCB will be in main storage, pointed to by level 3 of the ECB, and it will be in hold status.

Input Data

Input messages are presented to the application in a standard format described in the AM0SG DSECT or `c$am0sg.h` header. TPF performs the following processing before presenting a message to an application:

- Message elements (segments) are combined into an entire message.
- The elements are packed as one continuous character string in either a 381- or 1055-byte main storage block.
- When the entire message cannot be contained in a single 1055-byte main storage block, file pool storage is obtained, and subsequent message elements are packed into 1055-byte file storage blocks. These additional blocks are forward chained from the prime (main storage) block.
- For SNA systems, if the input message is defined as recoverable, the entire message is written to file pool storage for recovery purposes.

By including the AM0SG DSECT or `c$am0sg.h` header, the application program can address the fields in the message with the symbolic names in the header.

If the application requires you to use the older input message format (the MI0MI DSECT or the `c$mi0mi.h` header), it is the responsibility of the application input message editor to reformat the input message to that older format.

Table 3. Input Application Message Format: The `c$am0sg.h` Header

Data Type	**Displacement (Bytes)**	**Symbolic Name**	**Description**
`char`	0–1	`am0rid`	Record identification: MI for input
`unsigned char`	2	`am0rcc`	Record code check (not used)
`unsigned char`	3	`am0ctl`	Control byte
`char`	4–7	`am0pgm`	Filing program (for TPF use)
`unsigned long int`	8–11	`am0fch`	Forward chain - Must contain address of next file record (must be zero if no chain)
`unsigned long int`	12–15	`am0bch`	Last record in chain - must contain address of last record in chain (zero if no chain); used in prime record only, must be zero in all chain records
`short int`	16–17	`am0cct`	Character count from byte 18 to end of text (text + 5)

Table 3. Input Application Message Format: The `c$am0sg.h` Header (continued)

Data Type	Displacement (Bytes)	Symbolic Name	Description
`unsigned char`	18–20	`am0lit`	Origin address; sequence number, record identification, or LNIATA
`char`	21	`am0np1`	Control byte
`char`	22	`am0np2`	Control byte
`char`	23–N	`am0txt`	Text of message, including any function management header or terminal control characters

Note that `am0bch`, which you might expect to be a back chain, is not used as such by the TPF message router.

Application Message Editor

The first application program given control by TPF is usually called the input message editor or the application message editor, but the function is normally much broader than a data edit. The task varies with the complexity of the application, the variety of messages expected, and application design. Minimally, this program module must have an awareness of all the message types expected, exception conditions that may occur, and the logic to determine the appropriate program path. A good design is to make the input editor a component of a package of programs controlling all input and output message activity. The functions of this package would include:

- Editing and validation of the input
- Message integrity and recovery logic
- Program path selection for processing
- Maintenance of the terminal/conversational transaction control records
- Control of output messages.

This design is consistent with the top-down concept of structured programming. Figure 5 on page 45 shows an overview of the process flow for such a package.

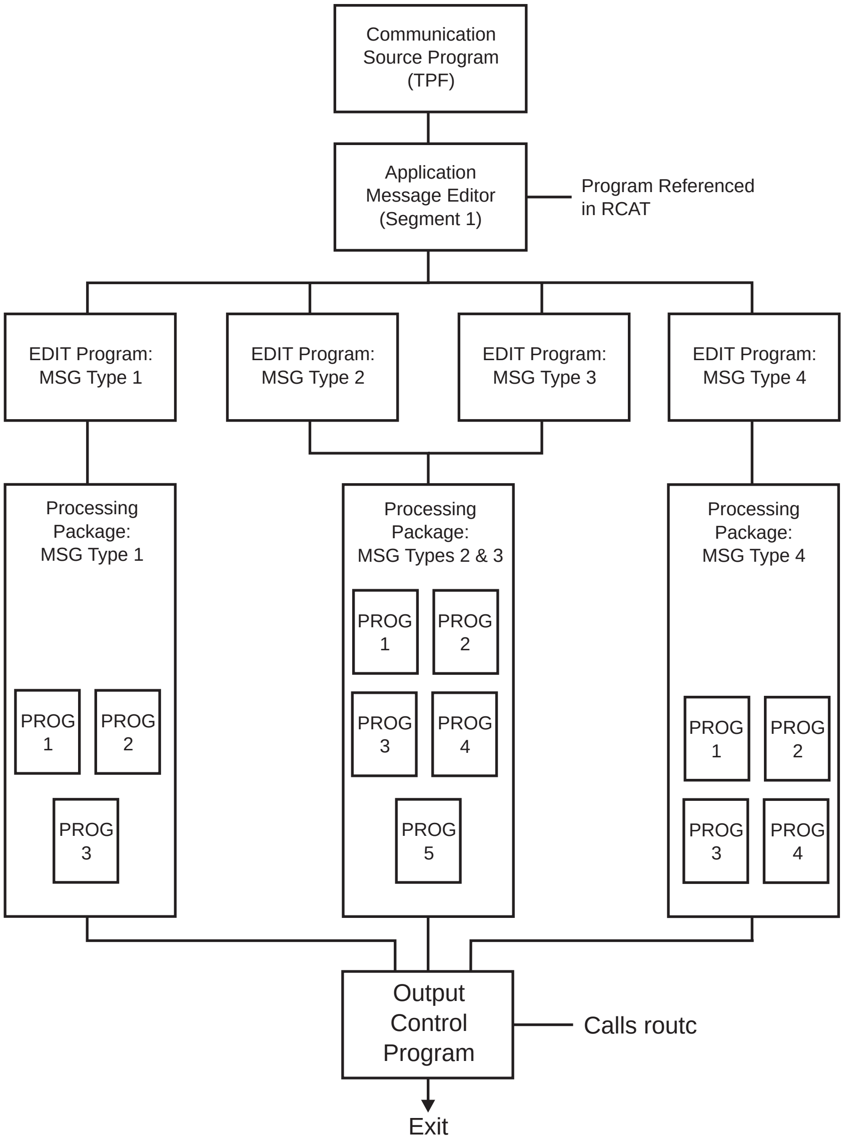


Figure 5. Application Message Control Package

Input Edit and Path Selection

As previously mentioned, the nature of the input and its variations will determine the type of edit required. Applications with limited message types may contain the complete edit in several program segments, which then pass control to the processing routines. When there are many message types or format variations, a more sophisticated approach will be necessary. For example, the text of the message may contain a specific character at a given position, which identifies its format and content. In this case, the editor could access a table such as Table 4, which matches action codes to editing program routines.

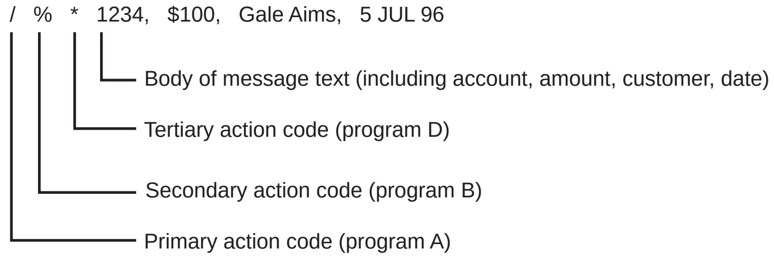
Table 4. Action Code Table (Example)

Action Code	Edit Routine
/	Edit routine A
%	Edit routine B

Table 4. Action Code Table (Example) (continued)

Action Code	Edit Routine
$	Edit routine C
*	Edit routine D
1	Edit routine E
2	Edit routine F

For variations within message types, additional action codes could be used to further define the expected format. Each message might contain, for example, primary, secondary, or tertiary action codes. Ideally, the action code is in the first position of message text. The edit program segment activated for a given primary action code can then test any additional codes by means of a subtable or whatever logic is most efficient for the expected message mix. Ultimately, each final routine in the edit process is aware of the appropriate program to call to continue processing. For example:


/ % * 1234, $100, Gale Aims, 5 JUL 96
Body of message text (including account, amount, customer, date)
Tertiary action code (program D)
Secondary action code (program B)
Primary action code (program A)

Message Recovery

The message editor program must be aware of the application message recovery options to be applied for this application. Message recovery is an optional feature of TPF SNA support, which varies with the type of system and the requirements of the application.

SNA Input Messages

For SNA systems, TPF maintains information on every SNA input message currently being processed. The level of recovery is user-selected, and it extends from full recovery of all input and output messages to simply keeping track of each input message in process. See *TPF ACF/SNA Data Communications Reference* for a complete description of this feature. For the host application programmer, the important aspect of message recovery is input recovery.

As each SNA input message is received, its origin and sequence number are recorded. The message is then passed to a TPF or a user-written transaction analysis routine to determine its input time-out interval and recoverability. Messages determined to be recoverable are written to file.

Note: Because of storage constraints in the routing control parameter list (RCPL), message recovery cannot be activated for 3600 multithread devices.

The input time-out interval is the time period TPF should wait for the application to respond. If no response is received before the time interval expires, the message is considered lost and is resubmitted to the application. The data resubmitted to the application is the original input if the input was recoverable or a canned message if the input was unrecoverable. It is critical that the application editor recognize this resubmitted input and supply the logic to complete the transaction or to cancel it. A bit in RCPL control word 0 (`RCPL2POS` in `rcplctl0`) identifies the message as returned or possible duplicate input.

SNA Output Messages

TPF SNA support also provides output message recovery. A message that cannot be delivered to its destination because the session between TPF and the logical unit has been lost is returned to the application with the returned message indicator bit (`RCPL0RET` in `rcplctl0`, the RCPL control byte 0) set. In addition, the origin and destination fields in the original outgoing RCPL are reversed when the message is returned.

The application may then:

- Reset `RCPL0RET` and send the message again; if the message still cannot be delivered, it will be returned again.
- Leave `RCPL0RET` set and send the message again; if the message still cannot be delivered it will be queued by TPF for later transmission.

When TPF returns an output message, the `RCPL0RET` bit is used to tell the application that it is a returned message. The application editor must recognize this returned output message and also must contain the logic to process the message according to its own requirements.

Note: If an application sends a message with a return request again, it must include appropriate code to prevent an infinite loop of returned messages.

An alternate method of handling returned output is for the application to place a unique application name associated only with returned output in the origin field of the output RCPL. This application name must be present in the RCAT in addition to the name for normal input. With this method, when a message is returned, TPF passes control to the editing program associated in the RCAT with the application name for returned messages. The point of this discussion is not to recommend a method but to mention the options, and to emphasize the application’s responsibility to implement this logic.

Application Recovery Package

TPF also supplies an application recovery package (ARP) that may be used by any system, SNA or non-SNA. The functions provided are similar in concept to SNA message recovery, but are designed to allow users maximum flexibility in its implementation. This package makes message recovery available for non-SNA systems, and it may be used by SNA systems as an alternative or supplement to support provided by TPF systems.

The ARP package is designed to allow the application to determine the extent of message recovery features, when they are performed, and the time-out factors. By using the application recovery table (ART), a unique reference is maintained for every active message in the system. At any point in the life of a message, the application can refer to the message and its control data (RCPL) and can file and retrieve the message block, or another related data block. The ART resides in main storage and contains the reference to the message plus an optional data area of up to 82 bytes, and the address of the data block on file. The filed block may or may not be a message block. For non-SNA systems, dependent on ARP alone for message recovery, the most common use is to file the message. SNA systems can depend on TPF to file the message block and can use ARP to file an additional data block such as financial totals affected by the transaction. Once the message is transmitted successfully, these totals can be considered secure and the filed block released. Should the message fail, the filed block can be retrieved and the affected amounts canceled.

The application passes control to ARP via an Enter to an external function call to an E-type assembler language program. In C language generally, the `TPF_regs struct` must be set up with a parameter list specifying the functions required before calling ARP. For TARGET(TPF) C language programs a `#pragma linkage` statement must be coded before calling ARP to allow the correct linkage to be made. Whether in assembler or in C language, although ARP can be used for output message recovery, for non-SNA systems the application must have a method of determining when the message is successfully delivered or has failed. This is an application responsibility; TPF does not supply it for non-SNA systems.

For details on the use of the Application Recovery Package, see *TPF Data Communications Services Reference*.

Terminal and Transaction Control

In the simplest configuration and application environment, each message has a uniquely identifiable terminal origin and is processed to completion without dependence on prior or subsequent messages. However, you probably do not experience this simplicity in practice. For example, you may need to build a transaction in conversational mode between a terminal operator, such as a bank teller, and application program that, perhaps, creates a new customer account. It would not be practical for the teller to enter all the data in 1 message. Rather, it should be entered in segments, consisting, for example, of name, address, telephone, initial deposit amount, other account data. This conversation requires a terminal operator-oriented record in which the status of the transaction can be stored after each message element and retrieved when the next element is entered. TPF provides the mechanism for this requirement with the following records:

- Routing control block
- Agent assembly area
- Scratchpad area.

The routing control block and agent assembly area predate SNA systems, although applications that use them may be accessed by SNA terminals. The scratchpad area is used only with SNA terminals.

Routing Control Block

There is one routing control block (RCB) record permanently assigned for every terminal addressed by a symbolic line, interchange, and terminal number (LN,IA,TA). The CPU ID further identifies the location of the terminal in the network. An RCB also is assigned to each symbolic line number used in a binary synchronous communication link.

When a message is received at the host from the terminal, TPF retrieves the RCB and passes it to the application message editor on data level 3. Because TPF assumes that the record may need to be modified, it is passed in hold status. While the RCB is in hold status, any further messages from the terminal associated with that RCB are queued. It is the application’s responsibility to unhold the RCB when it does not need to modify the RCB or lock out other messages from the terminal.

The RCB contains a system area, which should never be modified by the application, and about 700 bytes exclusively for application use. The application area is undefined and may be used by the application in any way. It could be used, for example, for building transactions in conversational mode, as outlined above.

Agent Assembly Area

Airline reservation systems often use an agent assembly area (AAA) instead of an RCB. The AAA is similar in concept to and uses the same addressing mechanism as the RCB. However, it was designed expressly for airline applications and does not have a reserved system area, nor is it initially passed in hold status. Any terminal capable of accessing an AAA-dependent application is actually assigned an AAA when it logs into any application.

Scratchpad Area

To provide support similar to the RCB for SNA systems, a system record and an application record are assigned to each logical unit. The system record is the node control block (NCB) and the application has no need to refer to it.

The scratchpad area (SPA) is allocated exclusively for application use, if desired. The SPA is undefined except for a 13-byte header. Your application should create a DSECT or define a structure that describes a record that suits your needs. The scratchpad area is allocated to fixed file storage and you can define it as 381 or 1055 bytes. The SPA (and NCB) records are ordered by resource identification (RID), which is in effect the network address of the logical unit.

Sometimes only a single terminal is associated with a logical unit, but in other cases, multiple terminals are associated with 1 logical unit (LU). In either case, however, the SPA records are allocated one per LU. This may not provide adequate flexibility or working storage. The application must design the record layout and a mechanism for additional file space per terminal that will meet its requirements. This will vary with the maximum number of terminals per LU, and the complexity of the application function. Possibilities include:

- The SPA could be allocated as a large record (1055 bytes) with logical sections for each terminal. Overflow could be chained to a random pool record in the same format. This solution would not be practical unless the number of terminals per LU were small (perhaps less than 10).
- The prime SPA could be formatted as above, with an overflow random pool record for each terminal, chained to the prime logical block.
- The prime SPA might contain common data usable by all terminals; an overflow random pool record could be chained for each terminal, as above.

SPA Retrieval

Since use of the SPA is optional, it is not automatically retrieved by TPF. However, a program is provided to retrieve it at your request. (See *TPF ACF/SNA Data Communications Reference*.) You pass control to the GETSPA program by calling the CSNB external function. For TARGET(TPF) C language programs a `#pragma linkage` statement must be coded before calling CSNB to allow the correct linkage to be made. In C language, the `TPF_regs` structure must be set up to load R1 with the address of a 5-byte parameter list, which is located on a halfword boundary. In assembler, before the ENTRC macro is invoked, R1 must be loaded with the address of a 5-byte parameter list located on a half word boundary.

You must define a structure or DSECT as follows to contain the required parameter list:

Size	Requirements
Bytes 0-2	Must contain the RID. This information is available from the first 3 bytes of the origin field of the input RCPL.
Byte 3	Must contain the data level for record (or file address) retrieval. The level may be expressed symbolically as D0–DF, or as its hexadecimal equivalent. For example:

Value	Data Level
X'00'	Data level 0
X'08'	Data level 1
X'10'	Data level 2
X'18'	Data level 3, and so on.

Size	Requirements
Byte 4	Specifies the user options as follows:

Bit	User Option
Bit 0=1	Always 1 for SPA retrieval. (This program is also used by system programs to retrieve the NCB record.)
Bit 1=0	Retrieve the record onto data level
Bit 1=1	Return file address only
Bit 2=0	Issue FIWHC assembler macro
Bit 2=1	Issue FINHC assembler macro
Bit 3=0	Return to user program if error
Bit 3=1	Exit if error
Bits 4–7	Not used.

On return to the application program, conditions will be specified in register R0 as follows:

Return Code	Condition
R0=0	Normal return.
R0=1	FACE error calculating ordinal number.
R0=2	Find error, `ce1sud` at the specified data level will further define the error.
R0=3	Error in input parameter list.

The SPA is filed using standard TPF file functions.

Function Management Header

TPF does not recognize the origin or destination of a message beyond the logical unit. It is the responsibility of the application to design a protocol between the application program in the cluster controller and the application program in the host that will identify (in the message text) multiple terminals associated with a logical unit.

One method of handling this is with the function management (FM) header, which may be included in the text of an input or output message. The FM header provides information about the text of the message. It may be used to indicate:

- The terminal associated with the logical unit that originated the message or is to receive the message
- The results of host processing, such as transaction status (completed, cancelled, or in process).

The format of an FM header is:

```
HL CI
```

where:

- HL is the header length, a 1-byte field indicating the length of the FM header including this byte.
- CI is control information, a user-supplied field, 1 to 254 bytes in length, providing information about the text of the message. The application designer is responsible for specifying the meaning and contents of this field.

When an FM header is provided, it must be the first part of the message text and its presence is indicated by the appropriate bit in the RCPL (in assembler RCPL2FMH=1 and in C language `RCPL2FMH` in `rcplctl2`, the RCPL control byte 2 in the `c$rc0pl.h` header).

The FM header may be maintained as received on input, modified or expanded, and included in the output. A unique RCPL bit indicates its inclusion in the output message (in assembler RCPL2FMO=1 and in C language `RCPL2FMO` in `rcplctl2`, the RCPL control byte 2).

Output Message Control

An efficient design for most applications is to concentrate the output message processing into a single program or package. Individual processing programs can pass requirements to the output processor usually with a single indicator or message number. Commonly used message texts can be formatted in a message file and referenced by number.

Whether handled by a single package or by the individual programs, the requirements for sending an output message consist of 3 steps:

Step 1 Prepare the data to be transmitted (as is described in the AM0SG DSECT or in the `c$am0sg.h` header) in a main storage block attached to any data level. If the entire message cannot be contained in a single block, the remainder must be placed in file pool records of the same size and chained from the prime main storage block. The DSECT or header for the AMSG record is the same for output or input: AM0SG DSECT or header `c$am0sg.h`. The symbolic names may be used by including the header. In assembler use the specified register with the CBRW of the data level chosen. In C use the `ecbptr` macro to point to the CBRW of the data level chosen. It is important that the chain fields in the record header be zeroed if not used; TPF will use the presence of a nonzero value in `am0fch` to indicate that an overflow file record exists.

Step 2 Construct the output RCPL, describing the destination and origin of the message. The output RCPL is similar in content to the input RCPL. It describes the origin, destination, and various indicators used for processing the output. Normally, the application preserves the input RCPL in order to construct the output RCPL. In most cases, the output RCPL can be constructed by swapping the origin and destination fields and setting the required indicators.

The destination field on output contains the symbolic network address of the terminal/logical unit to receive the message. For SNA systems it also contains the sequence number identifying the input to which this message is replying. This field, therefore, must be an exact copy of the input message origin field.

The origin field specifies the four-character name of the application sending the message. For SNA systems, when the message cannot be delivered to its destination, it will be returned to the application specified as the destination field of the input RCPL.

The application can enter a unique application name in the origin field on output so that undeliverable messages can be returned to that unique application. If so, the user must ensure that the alternate name is defined in the RCAT.

There are 4 or 8 bytes of control data, depending on whether the expanded RCPL is used. Sometimes only minimal changes to the input will suffice; in other cases, extensive resetting will be required. See the summary chart under “Routing Control Parameter List” on page 38 and, for more detail, to the RC0PL DSECT, the `c$rc0pl.h` header and to the *TPF ACF/SNA Data Communications Reference*.

Step 3 Request the TPF system to send the message by calling the ROUTC macro or the `routc` function, as described in *TPF General Macros* or *TPF C/C++ Language Support User’s Guide*. This is a simple coding routine once steps 1 and 2 are completed. The ROUTC macro or the `routc` function call specifies a pointer to the RC0PL DSECT or the `rc0pl` structure and the data level of the block in which the message resides.

Examples:

```
routc(&(ecbptr->ebw000),D6);

ROUTC LIST=R1,LEV=D6
```

Error conditions detected by the associated system service routine (`routc`) may cause a system error to be issued and the ECB exited. Error conditions include:
- Absence of a storage block at the specified CBRW
- Invalid message length
- Invalid contents of the message block and/or the RCPL.

Transmission of the message does not necessarily occur during the execution of the ROUTC system service routine, so the application may not be notified of transmission failure. However, when a detectable failure occurs (such as with SNA systems or applications able to use the application recovery package for output messages) the application specified in the RCPL origin field will be activated; the input RCPL will indicate this is a returned output message.

On return from the macro or the function, the CBRW indicates that the core block for the specified level is released to the system and no longer available to the application.

Table 5. Output Application Message Format: The `c$am0sg.h` Header

Data Type	**Displacement (Bytes)**	**Symbolic Name**	**Description**
`char`	0-1	`am0rid`	Record identification: OM for output (application must enter)
`unsigned char`	2	`am0rcc`	Record code check (not used)
`unsigned char`	3	`am0ctl`	Control byte
`char`	4-7	`am0pgm`	Filing program (for TPF use)
`unsigned long int`	8-11	`am0fch`	Forward chain - Must contain address of next file record (must be zero if no chain)
`unsigned long int`	12-15	`am0bch`	Last record in chain - must contain address of last record in chain (zero if no chain); used in prime record only, must be zero in all chain records
`short int`	16-17	`am0cct`	Character count from byte 18 to end of text (text + 5)
`unsigned char`	18-20	`am0lit`	Origin address; sequence number, record identification, or LNIATA
`char`	21	`am0np1`	Control byte
`char`	22	`am0np2`	Control byte
`char`	23-N	`am0txt`	Text of message, including any function management header or terminal control characters

Note that `am0bch`, which you might expect to be a back chain, is not used as such by the TPF message router.

TPF Macros

In the TPF environment, *macro* is a generic term covering several types of TPF user services. Generally, the application programmer is concerned with the use of a macro and not the details of defining the macro code, which becomes part of the MVS assembly system. For more information about macro definition, see *TPF Concepts and Structures*.

Understanding High-Level Language Concepts in the TPF System

This chapter introduces the terms and concepts used throughout this publication. Some of the terms describe features of C language and C++ language. See the user's guide, programmer's guide, and language reference for the IBM C or C++ compiler on the System/390 platform used by your installation for a more detailed understanding of these terms. See "IBM High-Level Language Books" on page xix for a list of IBM C and C++ compiler publications on the System/390 platform.

Other terms have special meaning in the TPF system environment. There is a glossary of TPF terms in the *TPF Library Guide*.

Note on Terminology

The use of the term *load module* supersedes that of *C load module*. Throughout this book, you will see C load module, which is a type of load module. For historical reasons, load modules on the TPF system were referred to as C load modules. The more generic term, load module, is used wherever possible so as not to be associated with a specific language.

See "E-Type Program Chart and Program Attributes" on page 56 for more information about the types of load modules.

Terminology is important to understand the concepts discussed in this chapter. The following is a list of some important terms:

Source file	This file consists of C, C++, or assembler language statements that define the actions of a program. A compiler or assembler reads this file as input.
Object file	A compiler or assembler output file that is suitable as input to a linkage editor. In the TPF system, object files are included in a load module as designated by the load module build script.
Load module	All or part of a computer program in a form that is suitable for loading into main storage for execution. A load module is usually the output of a linkage editor.
Dynamic link library (DLL)	A collection of one or more functions or variables gathered in a load module and executable or accessible from a separate DLL application load module.
DLL application	An application that can reference imported functions or imported variables in a DLL.
Mangling	The encoding during compilation of identifiers such as function and variable names to include type and scope information.
Bind	To combine one or more control sections or program modules into a single program module, resolving references between them, or to assign virtual storage addresses to external symbols.

Binder	The DFSMS/MVS program that processes the output of language translators and compilers into an executable program. It replaces the linkage editor and batch loader in the MVS/ESA or OS/390 operating system.

The word *program* is used generically to refer to processes and not just to the source file or executable load module.

The TPF system supports online programs written in C, C++, and System/390 assembler. Before its standard C support, the TPF system supported a nonstandard implementation of a C language subset known as TARGET(TPF) (from the compiler option that it required). TARGET(TPF) support is completely separate from standard C support and is no longer enhanced or used by the TPF system, although it is still available for legacy applications that have not been updated or migrated to standard C support.

In this publication, C or C++ refers to standard high-level language support. The term *TARGET(TPF)* is used to highlight significant differences or limitations between TARGET(TPF) and standard C support.

With the C or C++ language and compilers, you can create dynamic link libraries (DLLs) and DLL applications. See “C++ Support” on page 74 and “Dynamic Link Library (DLL) Support” on page 76 for more information about C++ and DLLs.

ISO-C and TARGET(TPF) functions can call each other; however, they operate in separate environments. This fact can have subtle effects where the global C environment is shared among C functions. For example, calling `setlocale` to change the current locale in a TARGET(TPF) function will have no effect on the locale for any ISO-C functions that may be called, and vice versa.

Note: TARGET(TPF) functions cannot call DLLs.

E-Type Program Chart and Program Attributes

Figure 6 on page 57 shows the kinds of E-type executable programs in the TPF system. A description of the attributes and characteristics of these programs follows the figure.

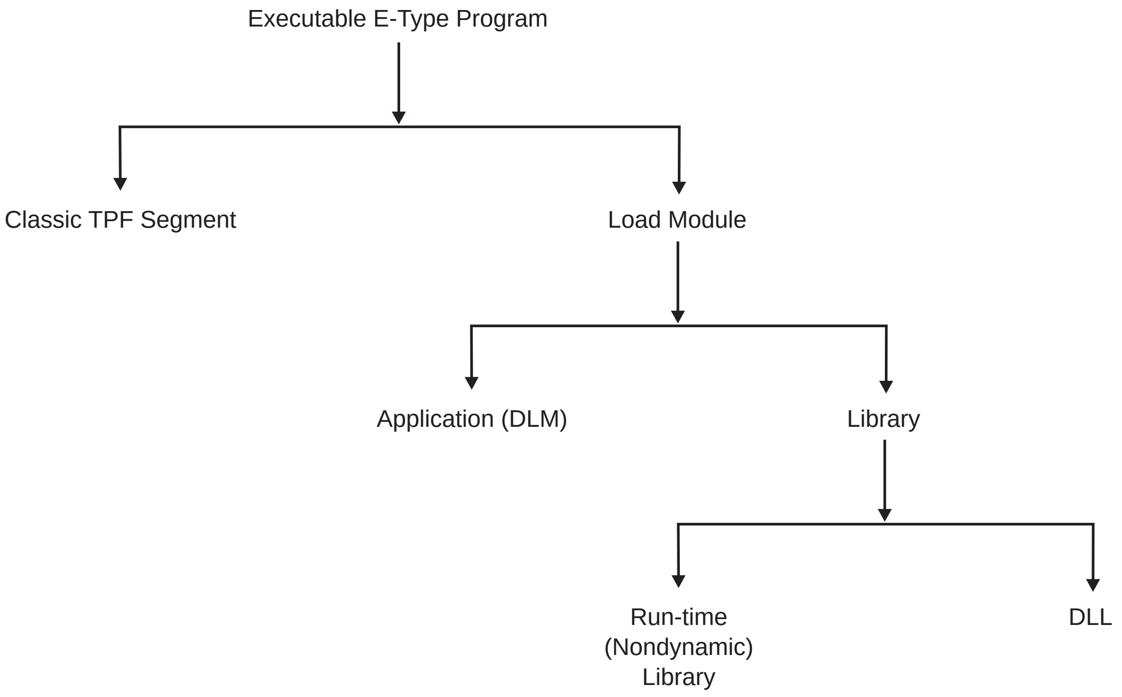


Figure 6. Kinds of E-Type Executable Programs

Load Module Attributes

This section describes some of the attributes of the various kinds of load modules. [1] Unlike *classic* TPF segments, load modules are link-edited (or bound). In general, load modules may include a variety of object files, which may be compiled or assembled from source code written in:

- Assembler
- C language compiled with the NODLL option
- C language compiled with the DLL option
- C++ language.

The *build script* of the load module specifies the type of load module (DLM, LIBRARY, or DLL) and the object files (but not the source language or compiler options) that the load module contains. The system initialization program (SIP) macro, SPPGML, specifies the source language and some of the compiler options, including RENT, NORENT, DLL, and NODLL, for TPF source code.

The primary distinction among the kinds of load modules is the kind of startup object file that is linked into them. The three main types of load modules are: applications (DLMs), run-time (nondynamic) libraries, and DLLs, each with its own separate startup object file. This determines many of their other attributes and ways that the TPF system handles them. The kind of load module is specified by the keyword in the first noncomment line in its build script. See “ISO-C Load Module Build Tool (CBLD)” on page 329 for more information about build scripts.

---

[1] There are other types of load modules in the TPF system, but here we are discussing only real-time load modules.

A secondary distinction concerns the name of the entry point function that is called first when an application (DLM) load module is run. This entry point function may either have the same name as the DLM or it may be named `main`. [2]

- DLMs that contain a `main` function are called by calling the `system` function in `stdlib.h`. This creates a new process and causes the calling process to wait for the new process to exit. If the initial `main` function call returns, the new process exits implicitly, or it can exit by calling a function or macro or taking a system error that causes it to exit. In any case, when the new process does exit, its exit status is returned to the calling process as the return value of the `system` function, and the calling process resumes running.
- DLMs that do not contain a `main` function (therefore their entry point function has the same name as the DLM) are called by being entered. If the initial entry point function call returns, there must be a program in the program nesting level to which control will return through BACKC linkage.

Load modules that contain object files written in C and compiled with the DLL option or written in C++ are *DLL applications* because they have the capability of implicitly referencing functions or variables that are defined in DLLs. Run-time (nondynamic) libraries are more restrictive than other kinds of load modules in that they cannot be DLL applications: they cannot contain object files compiled from C++ source code or from C source code compiled with the DLL option.

Table 6 lists the attributes of the various types of load modules.

Table 6. Load Modules and Their Attributes

Type of Load Module and Attribute	**Application (DLM)**		**Run-Time (Nondynamic) Library**	**DLL**
	With non-`main` Entry Point Function	**With `main` Entry Point Function**		
Keyword for the first noncomment line in the build script	DLM	DLM	LIBRARY	DLL
Name of the startup object file	CSTRTD	CSTRTD	CSTRTL	CSTDLL
Allocator residency requirements	File resident (FR), core resident (CR), or PRELOAD	FR, CR, or PRELOAD	PRELOAD	FR, CR, or PRELOAD
Number of entry points	one	one	one or more	one or more
Contains a `main` function	No	Yes	No	No
Specification of entry points	Function with same name as load module	`main` function	Specified by library interface script [1]	Specified by the EXPORTALL compiler option, `#pragma export` directive, or `_Export` keyword [2]
Name used by callers to call entry points	DLM name	DLM name	Names declared in API header files	Names declared in API header files
Linkage to entry points	C function call syntax, or enter- or create-type C function or assembler macro [3]	`system` function, declared in `stdlib.h`	Library call stubs, AUTOCALLed into load module by prelinker or binder	Resolved at run time either implicitly or explicitly

---

[2] A special case of DLM is an `iconv` translate table, which has its own unique entry point function naming convention.

Table 6. Load Modules and Their Attributes (continued)

Type of Load Module and Attribute	**Application (DLM)**		**Run-Time (Nondynamic) Library**	**DLL**
	With non-`main` Entry Point Function	**With `main` Entry Point Function**		
Linkage type of entry points	C (requires `extern "C"` declaration in C++ source code)	C (handled implicitly by the `system` function and TPF C run-time initialization code)	C (requires `extern "C"` declaration in C++ source code	C or C++
Can include assembler source code object files	Yes	Yes	Yes	Yes
Can include C source code object files compiled with the NODLL option	Yes	Yes	Yes	Yes
Can include C source code object files compiled with the DLL option	Yes	Yes	No	Yes
Can include C++ source code object files	Yes	Yes	No	Yes
Can call other executable E-type programs (using DLM call stubs)	Yes	Yes	Yes	Yes
Can call run-time library functions (using library call stubs)	Yes	Yes	Yes	Yes
Can explicitly access DLL functions and DLL variables using run-time library functions	Yes	Yes	Yes	Yes
Can implicitly call DLL functions and access DLL variables	Yes (requires C source code compiled with the DLL option, or C++ source code)	Yes (requires C source code compiled with the DLL option, or C++ source code)	No	Yes (requires C source code compiled with the DLL option, or C++ source code)
Can export DLL functions and variables	No	No	No	Yes

Notes:

[1] The interface to run-time (nondynamic) libraries, including the library call stubs, is built by the LIBI offline program. See “Run-Time (Nondynamic) Library Function Linkage” on page 65 for more information about run-time library linkage. For more information about the LIBI offline program, see “Library Interface Tool” on page 323.

[2] When a DLL is built, one of the prelinker or binder outputs is a definition side-deck, which contains information about the functions and variables that the DLL can export. When you build a DLL application, its build script specifies which definition side-decks are to be given as input to the prelinker or binder, which uses them to build the trigger functions that import DLL functions and variables.

[3] A load module can use C function call syntax to enter another executable E-type program (either classic TPF segment or DLM) with a DLM call stub. DLM call stubs are created by the STUB offline program or by SIP (if STUB=YES is coded on the SPPBLD macro in SPPGML) and AUTOCALLed into the load module by the prelinker or binder. See “DLM Call Stub Generator” on page 327 for more information about the STUB tool.

Classic TPF Segment

We use the term *classic* in this context to describe the TPF E-type segment that is written in basic assembler language (BAL) or TARGET(TPF) C language. Its characteristics are as follows:

- It is limited in size to 4 KB.
- If it is written in BAL, the output object module is ready to be processed by the TPF offline loader (TPFLDR) once it has been assembled by the high-level assembler (HLASM). To create an object module with HLASM, the source code must call the BEGIN (with TPFISOC=NO) and FINIS macros.
- If it is written in TARGET(TPF), the output object module is ready to be processed by TPFLDR once it has been compiled by a C compiler supported by the TPF system that also supports the TARGET(TPF) compiler option.
- It is not link-edited.

Functions and Calling Other Functions

The following tables show summaries of the calls that are supported in TPF E-type programs.

Table 7. Function or Program Calls Allowed in ISO-C

Type of Function (or Program) Originating the Call	**Dynamic Load Modules (DLMs)**	**DLMs with `main`**	**External Functions (Link Scope)**	**Library Functions**	**Static Functions (Source Scope)**	**Assembler Programs**	**Dynamic Link Library (DLL)**
Dynamic Load Modules (DLMs)	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes[7]	Yes[3]
DLMs with `main`	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes	Yes
External Functions (link scope)	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes	Yes[3]
Library Functions	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes	No[4]
Static Functions (source scope)	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes	Yes[3]
Assembler Programs	Yes[6]	No	Not Applicable	No	Not Applicable	Yes	No[5]
Dynamic Link Library	Yes	Yes	Yes[2]	Yes	Yes[1]	Yes	Yes

Table 7. Function or Program Calls Allowed in ISO-C (continued)

Type of Function (or Program) Originating the Call	Dynamic Load Modules (DLMs)	DLMs with `main`	External Functions (Link Scope)	Library Functions	Static Functions (Source Scope)	Assembler Programs	Dynamic Link Library (DLL)

Notes:

[1] means in the same object file.

[2] means in the same load module.

[3] means the function or program originating the call must be compiled with the DLL compiler option.

[4] Library functions cannot be compiled with the DLL option; therefore, a run-time library function cannot call a DLL.

[5] A segment written in assembler cannot call functions in a DLL.

[6] A segment written in assembler can call the entry point function in a C++ load module because of the `extern "C"` wrapper for the entry point function. This wrapper makes the linkage to the entry point of the load module be C linkage.

[7] If you want to call a program written in assembler from a program written in C++, you need to put the C function prolog and epilog macros, TMSPC and TMSEC, around the program written in assembler. You also need to prototype the program written in assembler as a C function.

Note: The parameter list structure and linkage are different between C and C++. Therefore, functions coded in C and compiled with a C compiler cannot call functions written in C++ unless the C++ function is declared to have C-type linkage using `extern "C"`.

See Table 6 on page 58 and "`main` or `extern "C"` Requirement" on page 140 for more information about this linkage difference.

Table 8. Function or Program Calls Allowed in TARGET(TPF).

Type of Function (or Program) Originating the Call	External Functions	Library Functions	Static Functions	Assembler Programs
External Functions	Yes	Yes	Yes *	Yes
Library Functions	Yes	Yes	Yes *	No
Static Functions	Yes	Yes	Yes *	Yes
Assembler Programs	Yes	No	No	Yes

Note: * means in the same module.

There are several different types of items that DLMs can call using function call syntax:

- BAL segments
- All other C functions, including inline, internal, external, library, TARGET(TPF), and other DLMs
- Exported functions in DLLs.

The only difference between calling BAL segments and other C functions is that the function prototype for BAL segments is restricted. If you call a BAL segment, the prototype must be:

```
void SEGN(struct TPF_regs *);
```

where SEGN is the 4-character segment name.

The call to a BAL segment looks like:

```
struct TPF_regs regs;
/* Set up regs as appropriate to the BAL segment interface */
SEGN(&regs);
```

In standard C or C++, if the BAL segment does not use registers in its interface, you can code a NULL pointer:

```
SEGN(NULL); /* You MUST code the regs parameter for BAL calls,  */
            /* even if the called segment doesn't use registers */
            /* for its interface.                               */
```

Do not use the NULL pointer instead of TPF registers in TARGET(TPF).

Functions Calling Other Programs

There are several functions that activate only E-type programs. The following functions call E-type programs:

`entdc`	reset stack, no return (drop nesting levels)
`credc`	create new deferred ECB
`creec`	create new ECB with core block attached
`cremc`	create new immediate ECB
`cretc`	create new time-initiated ECB
`cretc_level`	create new time-initiated ECB with core block
`crexc`	create new low-priority deferred ECB
`crosc_entrc`	cross-subsystem call with return
`swisc_create`	create new ECB specifying I-stream
`tpf_cresc`	create new synchronous ECB
`system`	execute a command

If a function is written in assembler with the TMSPC and TMSEC macros, this assembler function can activate E-type programs. The following is a list of the BAL equivalents of the C macros:

ENTDC	enter dropping nesting levels (reset ISO-C stack)
ENTRC	enter with return
CREDC	create new deferred ECB
CREEC	create new ECB with core block attached
CREMC	create new immediate ECB
CRESC	create new synchronous ECBs
CRETC	create new time-initiated ECB
CREXC	create new low-priority deferred ECB
CROSC ENTDC	cross-subsystem ENTDC
CROSC ENTRC	cross-subsystem ENTRC
SWISC ENTER	cross-I-stream enter (ENTDC)
SWISC CREATE	create new ECB specifying I-stream

Notes:

1. Load modules do not use ENTNC and CROSC ENTNC macros.
2. For TARGET(TPF), writable static storage is deallocated by `entdc`. For standard C, writable static storage is preserved across `entdc`.

ISO-C Linkage Performance Considerations

How ISO-C functions are packaged can affect the overall performance of a program. There are several ways to package functions:

- As inline functions of a C or C++ compiler. These functions are directly in place where they are called.
- As internal functions added by the linkage editor. These functions are contained in the same load module as the calling function.
- As external functions contained in a DLL. These are function calls to separately compiled programs. See “Dynamic Link Library (DLL) Support” on page 76 for more information about DLLs and “C++ Support” on page 74 for more information about C++ support.
- As library functions. These functions are referenced by stubs added to the load module during linkage editing.
- As external functions in the form of a DLM. These are function calls to separately compiled programs.

There are advantages and disadvantages to each packaging technique.

INLINE Compiler Option

The fastest performance for a function comes from using the `INLINE` option of the compiler. Using inline functions eliminates entirely call linkage, function prolog, function epilog and return linkage, and allows global optimization of calculating function arguments and the function body. If simple functions must be called many times in a loop, the combination of inlining and global optimization can yield significant improvements in path length.

Note: Because there are no references to certain functions with inlining, there are no pointers to them. These functions will not be part of the link map for the load module. See *TPF Operations* for more information about the ZDMAP command and link map support.

One cost of using inline functions is maintenance. The source code for the functions is included in header files for all the programs that use them. If a function needs to be updated all applications that call it must be recompiled, relinked, and reloaded.

Other disadvantages to inline code are:

- Functions must be defined in the same compile unit in which they are inlined.
- Code size increases when non-trivial functions are inlined multiple times.
- Debugging can be more difficult because inlined functions do not have a single entry or exit point, and may generate different object code each time they are inlined.

Inlining is an optional feature of the IBM C compilers on the System/390 platform. Inlining is standard for the IBM C++ compilers on the System/390 platform. It is not part of the ANSI/ISO C standard.

Linking Functions

The next fastest performance is determined by how functions are accessed, whether through a stub or not. Functions that are accessed without using a stub are faster than those accessed through a stub.

For function calls that do not need a stub, the compiler, linkage editor, and relocating loader (program fetch) generate the linkage, which is executed without use of system services or system data. Call linkage is usually 2 instructions, plus parameter loading and unloading, and return linkage is included in the function epilog.

The cost of a function in the same compiled unit as its caller and the cost of link editing functions together are much the same. It is a tradeoff between program execution and easier maintenance. If a function needs to be updated, only the function itself must be recompiled, but all applications that use the function still must be relinked, reloaded, and tested again.

Using Library Functions

The use of library functions involves a slightly longer path length than the use of link-edited functions. This is because the calling program must use a library call stub to access library functions. In addition, secondary linkage processing makes the path length for the library calls much longer if one of the following conditions exists:

- The library load module contains static data.
- User exits are active.
- The library was selectively activated using the E-type loader.

The advantage of library functions is that they are easy to maintain and reuse in many applications. If the function needs to be updated only the function needs to be recompiled, and only the library load module needs to be relinked and loaded.

A disadvantage is that at a given time, for a given library, all functions use the same linkage. Libraries that contain reentrant (RENT) static ALWAYS use secondary linkage. Therefore, library functions that require reentrant static should not be included in libraries that must give high performance.

DLM Performance

DLMs are the primary structure for ISO-C functions. Call and return linkages between DLMs are managed by TPF system services and are comparable to ENTRC and BACKC linkage. There is additional overhead for managing linkages between different types of programs.

The performance decision involves the number of DLMs that need to be created for a given application. The fewer DLM calls needed, the faster a given application performs. Calls to DLMs generate the most linkage overhead because they:

- Utilize enter/back services
- Create additional stack frames
- Manipulate static storage pointers.

The disadvantage is the loss of visibility of the function of a DLM to the system. If DLM A is folded into DLM B to improve DLM B’s performance, the service that DLM A provides is lost to other DLMs (without replicating DLM A).

Run-Time (Nondynamic) Library Function Linkage

Note: This section discusses library function linkage for run-time (nondynamic) libraries only. See "Dynamic Link Library (DLL) Support" on page 76 for more information about DLL linkage.

ISO-C uses an array of addresses to library vectors (AOLA) to provide its primary linkage to library functions. See Figure 7. The AOLA is created during TPF restart. AOLA entries are replaced with pointers to library vectors (LIBVECs) as libraries are read into main storage. A LIBVEC is a series of entry point addresses of functions.

Each library is associated with a particular library ordinal, defined by the library interface tool. This library ordinal defines the order that the libraries are placed in the AOLA. See Table 9 to see which ordinals are assigned to which libraries. The AOLA mechanism allows calls to functions in as many as 1024 libraries. A library ordinal in the AOLA and a LIBVEC offset together serve to uniquely identify a particular function.

When a new version of a library included in an E-type loader loadset is activated, a common block is used to build a new AOLA. The common block is key-protected.

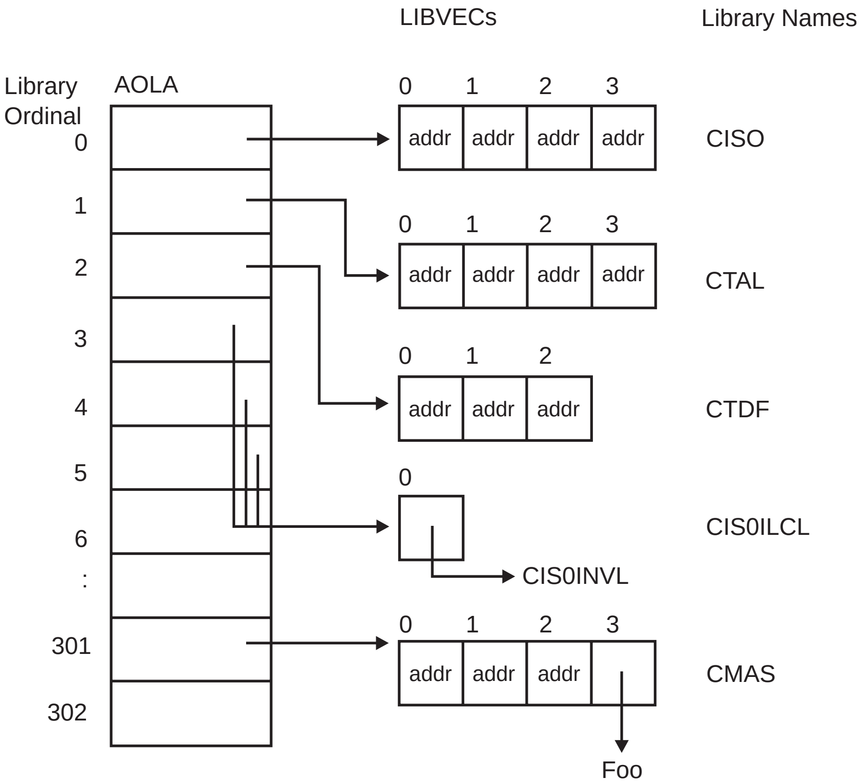


Figure 7. Example of an Array of Library Addresses and LIBVECs

Table 9 describes which ordinals are in use by which library.

Table 9. Library Ordinals

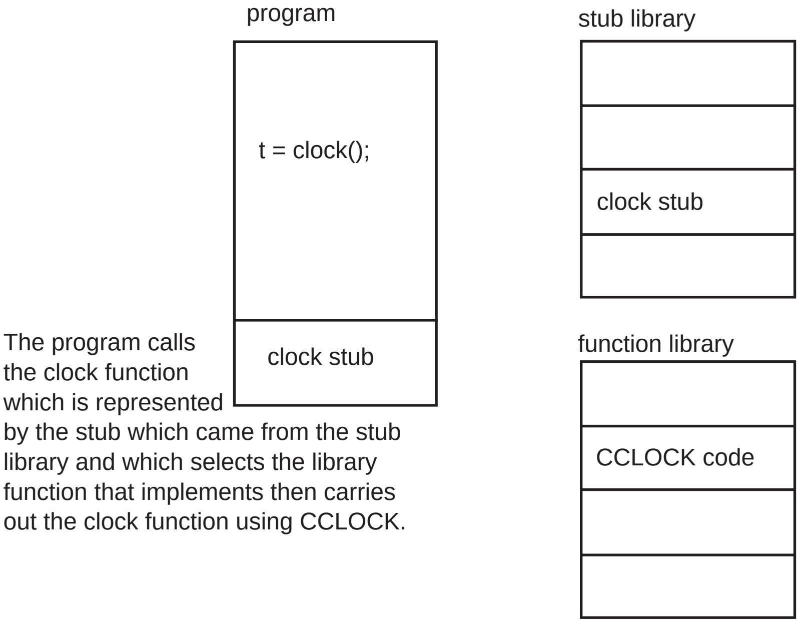
Library	Ordinal Number	Description
CISO	0000	Standard C library
CTAL	0001	TPF API library

Table 9. Library Ordinals (continued)

Library	Ordinal Number	Description
CTDF	0002	TPFDF library
CTBX	0003	ISO-C general purpose toolbox library
COMX	0004	TPF communications functions
CMQI	0007	MQI client library
CTHD	0008	threads library
CRPC	0011	RPC library

Function Stubs

Linkage to functions contained in libraries is through a *stub*. A stub is a small piece of code on the end of a program used to locate functions in libraries. For example, the `clock` function is contained in the TPF library CISO. Imagine a program BAZ calls `clock`. To gain access to the `clock` function, control transfers to the stub for `clock` on the end of BAZ. The stub was added during the link-editing of BAZ. The stub itself comes from the stub library created by the library interface tool. The stub for `clock` contains the library ordinal and LIBVEC offset for the actual `clock` function. The executable form of the `clock` function was loaded during TPF restart when the CISO library was loaded. The information in the stub provides run-time access to the code for `clock` and the stub transfers control to the `clock` code.



The program calls the clock function which is represented by the stub which came from the stub library and which selects the library function that implements then carries out the clock function using CCLOCK.

Suppose we have a program BAZ. Overall, when the C compiler encounters an external function call (Foo) in an ISO-C program, it generates external function linkage. While satisfying the external linkage, the linkage editor gets a stub to access the external function. (See the Figure 8.)

The address of the function required is not available at compile time. The compiler generates a VCON to be satisfied during linkage editing. During linkage editing the VCON for the external function is satisfied by using the stub function that corresponds to the function called. The linkage editor retrieves the stub from the stub library and appends it on the end of the load module being generated. The stub contains the library ordinal and the LIBVEC offset. When the stub was generated, it was set with information about the location of the function it represents. In the example function Foo is the fourth function defined in library CMAS (301), so the library ordinal is 301 and the LIBVEC offset is 3. When run, the stub function performs like an execution-time ″glue module″, sticking the library references kept in the function body together with the library as it exists in the online system.

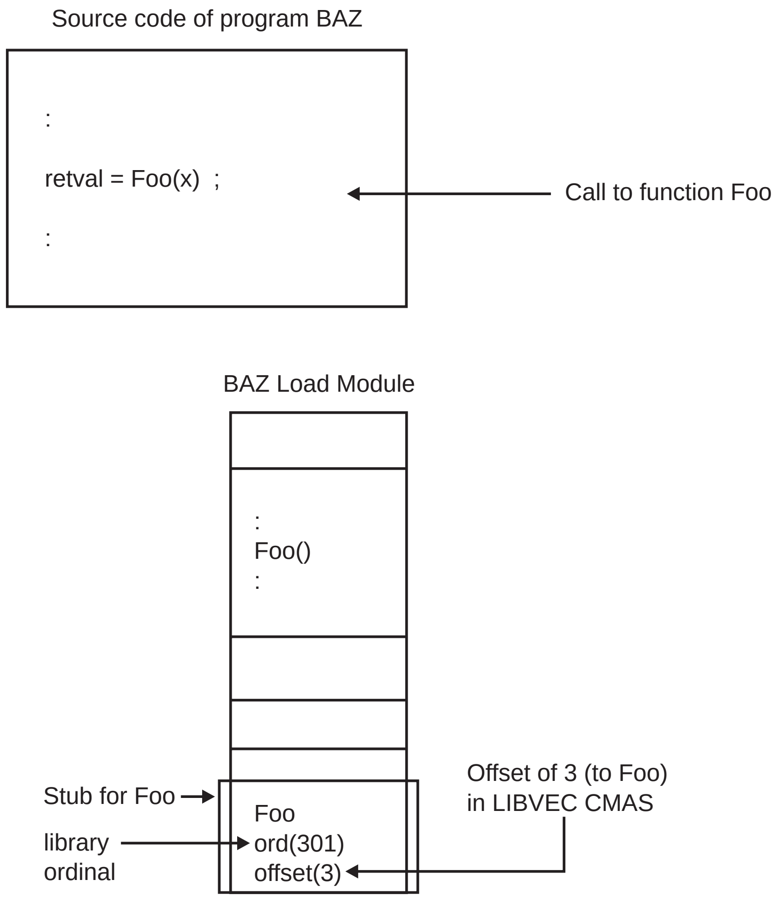


Figure 8. Offline Stub Linkage

When segment BAZ runs on the online system and finds a call to function Foo, the stub for Foo at the end of BAZ uses the library information to locate the executable code for Foo. (See Figure 9) The stub does this by using the library ordinal (301) it contains to look up the pointer to the correct LIBVEC in the AOLA. Using the LIBVEC pointer and the offset (3) it picks up a pointer to the code for Foo (in the CMAS user library). With this LIBVEC the stub uses the function offset to locate the entry point address for function Foo, which was created when the library with Foo was loaded (during restart). The stub transfers to the entry point address to run the Foo function. When Foo returns, it continues at the NSI after the original call to Foo in program BAZ.

It is critical that the order of the libraries in the AOLA must correspond to the order of the libraries used when programs are compiled and linked. Clearly, if the order of the 2 libraries is different, the functions called by the program will not be the functions run by the system.

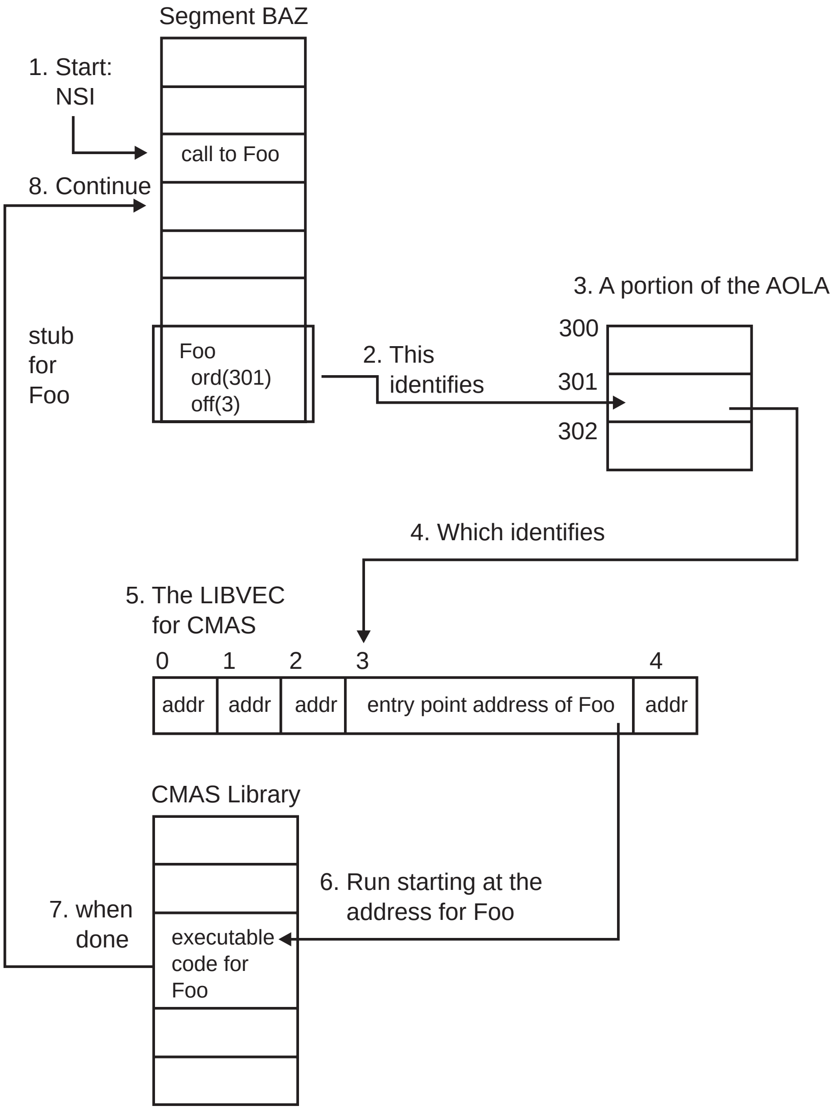


Figure 9. Online Function Linkage

Secondary Linkage for ISO-C Support

If a library requires writable static, or contains an active user exit, or is selectively activated using the E-type loader, the linkage for any function in the library uses secondary linkage. In this case, the AOLA entry for the library is a pointer to the secondary LIBVEC. The LIBVEC itself points to a branch vector that receives control instead of the function directly. When the function is called, common code handles the static storage pointers or activates the user exit routine. This additional overhead causes slower performance in these functions.

Writable static involves such overhead that you should be clear about what it is. In C language a variable can be declared static. This means the variable is not reinitialized every time the function containing it is called. An example of static data is a constant character string (like ″this is a test″). Constant data can be reinitialized with every function call without loss. What if you wanted to prevent the data from being initialized every time? If the static character string contained a part that changed (like ″this is Bob″, ″this is Fred″, and so on), something special would be required to keep the information. This something special is secondary linkage. Secondary linkage is required to keep track of Bob one time, Fred the next, and so on.

Secondary linkage is described in Figure 10 following. Function Foo, which is found in segment BAZ, has its associated library information. The stub for Foo, which is the same whether primary or secondary linkage is used, is at the end of BAZ and uses the library information. The AOLA entry either points to the executable code for Foo in the CMAS library (primary linkage) or it points to the library startup code at the beginning of the library (secondary linkage). Every library has library startup code at its beginning. The startup code takes care of some housekeeping chores before and after transferring to function Foo. If the CMAS library contains writable static or has the user exits active, secondary linkage is used instead of primary linkage.

In the figure primary linkage is shown as a simple double arrow and secondary linkage is shown as steps A, B, and C.

If writable static is required, space in the frame is set up to accommodate the static. If the user exits are active, they receive control. When the user exit ends, the startup code transfers to the entry point address of Foo. When Foo ends, it returns to the startup code, which transfers to another user exit, if they are active. When this ends, the startup code returns to segment BAZ which continues with the next sequential instruction after the call to Foo.

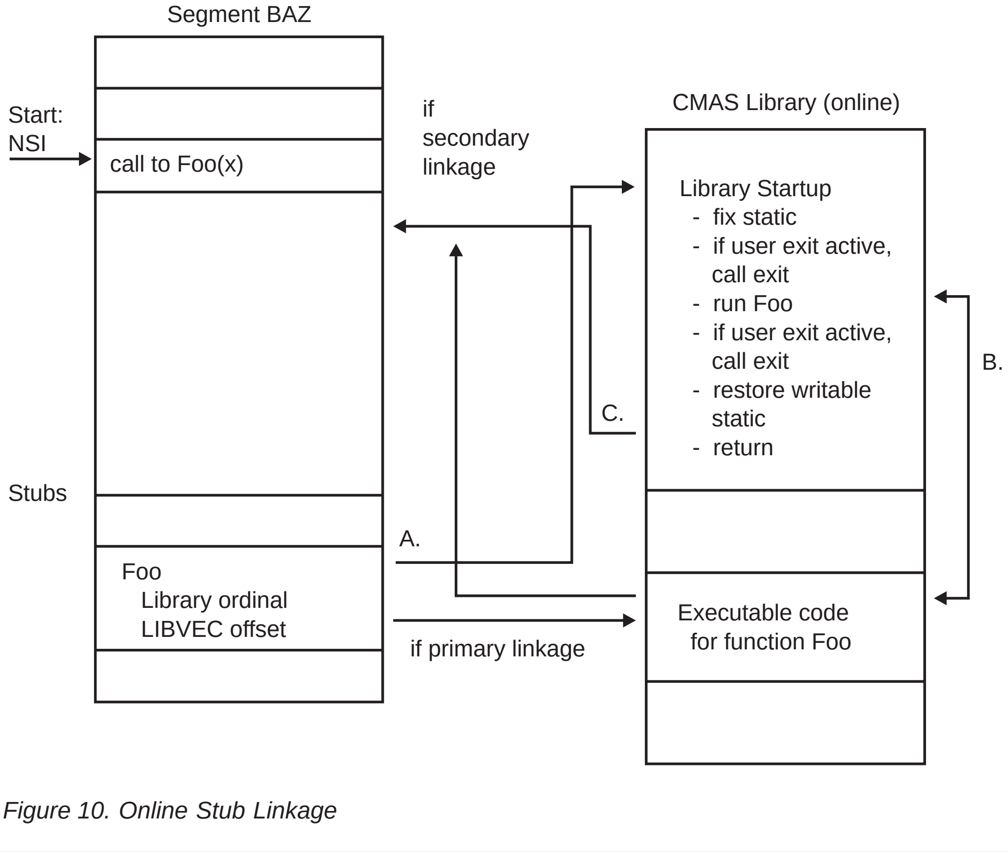


Figure 10. Online Stub Linkage

Quick Enter Directory for TARGET(TPF)

Note: The quick enter directory does ***not*** exist for ISO-C support.

In TARGET(TPF) library functions are implemented as a collection of common programs in main storage. The location in memory of all the library functions is maintained in the quick enter directory, which is built during system restart. (The quick enter directory is also known as the *primary directory).* The basis for the information in this table is provided at installation time by way of *CLIBFUN* macro calls, which are assembled into program C000.

For more information about the CLIBFUN macro, see “Customizing C/C++ Language Support” on page 301.

Secondary Directory for TARGET(TPF)

Note: The secondary directory does ***not*** exist for ISO-C support.

Some of the C library functions (such as the math functions) are coded in assembler and do not follow the conventions required for TPF ECB-controlled programs. We refer to these as *secondary library routines.* These routines reside in their own control program CSECTs. When a function that corresponds to one of these secondary library routines is called, a program is activated that generates linkage to the TPF service routine. The service routine transfers control to the appropriate CSECT and obtains the address where the secondary library routine resides from a table of VCONs known as the *secondary directory*.

Storage for Static Variables

Static storage contains the static variables declared in a C language program. For ISO-C the static storage is unique to a load module, subsystem, and environment. There is only 1 static storage allocation for each load module in the same subsystem and environment per ECB. The storage block used for static storage is obtained by the DLM startup code or library startup code that requires it. When static storage is required, the static storage block for the load module is found if one exists or obtained through a call to the $GMNBC macro if one does not exist. This storage comes from the ECB heap.

When writable static variables are used in a function, static storage maintains the values of these variables. To handle the static storage in libraries, secondary linkage is employed when a function is run. The secondary LIBVEC is used to manage processing the writable static. The extra overhead of this processing can cause a performance loss.

In TARGET(TPF) a chain of working storage blocks is used to contain the storage needed for any static variables declared in the C source module. These blocks are known as *static blocks* and contain 1 or more *static frames*. Each static frame maps to a single TPF C program segment.

In standard C, once a static frame is allocated, it remains in existence until the ECB exits. The same is true for TARGET(TPF) except for the case where `entdc` is called. In TARGET(TPF), the static block is removed if the ECB called an `entdc`.

In general the static exception routine handles the process of allocating static blocks.

For more information about coding with static storage, see “Static Storage Considerations” on page 143.

ISO-C Language Stack

In ISO-C, a stack is used to manage function environments represented by local variables and parameter lists. The stack is allocated from a contiguous virtual address space set aside for the ISO-C stack. The first time a C function is called, an initial storage area (ISA) is set up to handle the C environment and control blocks. It contains a work space (LWS) that can be used by the functions managed by the stack. If the called function calls another function, a new frame is added to the stack. Each frame holds space for information relevant to a given function (such as the automatic variables and a register save area). When the function returns, its frame is removed from the stack and the frame for its caller becomes the current frame.

The largest amount of stack storage an ECB can acquire and the amount of each stack increment are determined by fields in keypoint A. These fields are copied to an ECB page during restart. The amount of the stack increment for an ECB can be adjusted using the ZCTKA ALTER command and in the ECB creation user exit (see UCCECB in *TPF System Installation Support Reference*). If an ECB exceeds the amount of stack storage initially carved, an overflow routine is called to extend the overall stack size by a stack increment or the current required size. Stack extension is done in multiples of 4KB. If the ECB tries to acquire stack storage greater than the maximum allowed for it, a system error results and the ECB exits.

See ICS0TK DSECT for a detailed layout of the TARGET(TPF) stack frame and the DSECTs IDSDSA and IDSLWS for the ISO-C stack frame layout.

TPF Header Files

An integral component of C language is the ability to include function prototypes, parameter definitions, preprocessor directives, and commonly used structures in what is known as a *header* or *header file*. The preprocessor `#include` directive is used to specify that the contents of a particular header file are to be included as part of the C source module.

The TPF system provides several header files that support the TPF API functions and map common TPF data structures and the IBM released *globals* (frequently used and commonly accessed symbols and values that reside in main storage). For example, header file `c$eb0eb.h` maps the ECB structure. In addition, TPF provides a subset of the C/370 header files for use when compiling C code for the TPF system.

Once ISO-C support has been added to your system, if a TARGET(TPF) program is to be compiled, the DEFINE(_TARGET_TPF) compiler option must be specified. DEFINE(_TARGET_TPF) can either be specified as a command line parameter to the compiler or it can be added to the source code as **#DEFINE _TARGET_TPF 1**. Header files and macros will produce ISO-C compatible code unless _TARGET_TPF is defined. For a list of these header files, see “Customizing C/C++ Language Support” on page 301.

Note: See “TPF Header Files and C++” on page 145 for information about the changes to the header file structure of C programs for C++.

Run-Time Libraries

Note: The following describes run-time (nondynamic) libraries only. See "Dynamic Link Library (DLL) Support" on page 76 for more information about DLLs.

Support for multiple libraries of functions is supported. These libraries must reside in main storage. The IBM-shipped libraries are the most widely used libraries in the complex. They are allocated as PRELOAD and loaded during restart. Figure 11 shows an example of a library load module. If a library is not loaded during restart, the online system cannot use it.

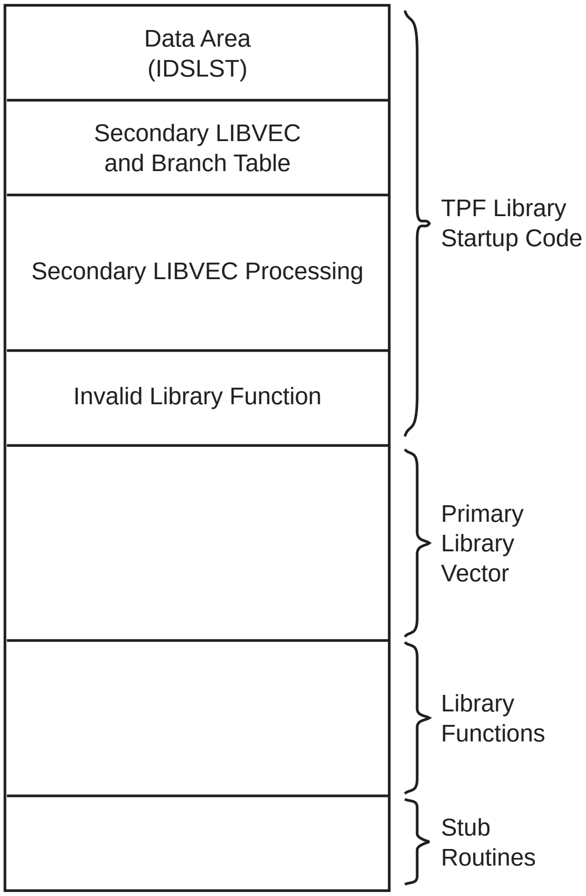


Figure 11. Components of an ISO-C Library Load Module

The TPF library startup code contains data about the library load module and code that processes the secondary library linkage. IDSLST DSECT is used to map the data portion of the startup code.

All libraries should be allocated as core resident, PRELOAD, shared, and with a 31-bit addressing mode. These attributes are assumed by the loader regardless of how the libraries are allocated. Moreover, ISO-C programs cannot be allocated as private or as I-stream unique.

If a new library is being created and loaded with the E-type loader, a new program name is needed. It is not possible to load a new version of an existing program that contains the new library unless the existing version is a library.

Dynamic Load Modules

Dynamic load modules (DLMs) are compiled and linked source programs that are ready to be loaded into main storage and run. Although a DLM has a single entry point, it can consist of many parts or subfunctions. The DLM is created by linking together separately compiled object files into a single load module.

A DLM transfers control back to its caller from its entry point function. Control can be given up either explicitly with a `return` or `exit` statement or implicitly when the last line of the function runs.

The name of a DLM entry point must match the 4-character DLM name or `main`. Use prototype statements to define the valid parameters for a DLM.

Note: If the entry point function is written in C++, it must use `extern "C"` linkage.

Creating Globals for C

TPF globals are normally accessed using assembler labels, which are not accessible to the C or C++ compiler. Therefore, a set of C global tags, known as *tag names*, has been created that correspond to the assembler labels. (TPF provides a sample utility, called *GNTAGH* which uses the ADATA generated by the HLASM assembler as input to show how this process might be automated.) Each global tag name maps to a unique 32-bit value that describes the displacement of the item in its global area, its length, and other attributes. These tag names make up the `c$globz.h` header file. Header file `tpfglbl.h` contains TPF-defined global constants and function prototypes.

See “Customizing C/C++ Language Support” on page 301 for more information about TPF global tags. See *TPF C/C++ Language Support User’s Guide* for detailed information about the GENTAG program and creating C global tag names.

C Language Locale

There is a *locale* associated with each C environment that consists of a set of constants that vary with geographic area, such as time zones and monetary symbols. Locales are often associated with particular countries, such as the USA or France. The `setlocale` function can be used to specify a particular locale.

In ISO-C, the default locale is defined in the ISO-C library (CISO) and resides in segment C$S370. The C locale is the first one set by the system; the other locales reside in separate DLMs. For TARGET(TPF) the locale resides in segment CL04, a part of CCLANG, the control program CSECT for C language.

In TARGET(TPF), pointers to locale information are stored in the first stack frame associated with an ECB. This stack frame is created and initialized the first time a C program or function is called (when the stack exception routine is invoked). A user exit is provided that allows you to specify the particular locale definitions you want to use.

For more information about defining locales, see “Customizing C/C++ Language Support” on page 301. For more information about the C stack exception routine user exit and others, see *TPF System Installation Support Reference*.

Character Set Support

A number of offline tasks must be completed to create support for a new character set on the online system. For information on character sets, see “Character Sets” on page 314.

C++ Support

C++ language introduces object-oriented (OO) concepts into C language. Built on a foundation of C language, C++ adds support for OO programming along with many other features. Because it is based on C, much of what you know about C applies to C++.

Class Library Support

The TPF system provides class library support for the following:

- The I/O Stream Class Library, which provides facilities for handling many varieties of I/O such as `cin`, `cout`, `clog`, and `cerr`. See *OS/390 C/C++ IBM Open Class Library Reference* for more information about this class library.
- A subset of the C++ classes of the Application Support Class Library. The classes provided are:

  IBinaryCodedDecimal
  The `IBinaryCodedDecimal` and `decimal` classes allow you to represent numerical quantities accurately in business and commercial applications for financial application.

  IDate The `IDate` class provides support for date information. You can construct `IDate` objects in a number of ways and then use `IDate` methods to determine the day of the week, month, or year, compare two dates, test a date for certain characteristics, and obtain the names of days or months that are dependent on the national language locale setting in effect at run time.

  IException
  The `IException` class is the base class from which all exception objects thrown in the library are derived.

  IString
  The `IString` class provides greater flexibility in handling strings than traditional C-style character arrays. The `IString` class supports both single- and multiple-byte character sets. With `IString` objects, you can code string-handling operations much more quickly. For example, you can concatenate two strings simply by using the +operator, or compare them using the == operator.

  ITime The `ITime` class creates time-of-day objects. You can compare the objects, add them together, remove specific information from them, or write them to an output stream.

  ITimeStamp
  The `ITimeStamp` class creates time-stamp objects. You can compare the objects, add them together, remove specific information from them, or write them to an output stream.

  I0String
  The `I0String` class is identical to the `IString` class except in its method of indexing strings. In the `IString` class, the first character of a string is at position 1, whereas the same string when stored in an `I0String` object

has its first character at position 0. `I0String` is provided for programmers who are used to the C string-handling approach of treating strings as starting at position 0. `IString` and `I0String` objects are easily interchanged, and they support the same set of methods and operators.

Note: When you use the objects of the exception classes in the Application Support Class Library, the following series of messages may be displayed:

```
Error ID is nnnn
Error Code group is string
Exception Text is:
  Refer to Class Library Support in the TPF Application Programming book.
```

where *nnnn* is the error number and *string* is the group associated with the error ID. See the *OS/390 C/C++ IBM Open Class Library Reference* for more information about this series of messages.

TPF message OPR-I094204 may also be displayed immediately following these messages. This message is the result of the same error condition, so you can ignore it.

- As described in *OS/390 C/C++ IBM Open Class Library Reference*, the following header files are used by applications to make use of the Application Support Class Library (CPP3):
  - `idate.hpp`
  - `idecimal.hpp`
  - `iexcbase.hpp`
  - `istring.hpp`
  - `itime.hpp`
  - `itmstamp.hpp`
  - `i0string.hpp`.

  All other header files shipped with the Application Support Class Library are for implementation only.
- The STLport standard template library, which is a standard template library that contains generic container classes and algorithms. The *container classes* are used as templates to define objects, while the *algorithms* are used to manage data in the containers. STLport standard template library code, which is available on the Web at http://www.stlport.org is ***not*** shipped with the TPF system, but you can port this code to the TPF system. For more information about porting STLport code:
  1. Go to the TPF Web page at: http://www.ibm.com/tpf/.
  2. Click **Site map**.
  3. Click **STLport**.

See the *OS/390 C/C++ IBM Open Class Library User's Guide* for more information about the C++ classes.

TPF Restriction

If you code a `throw` block in a destructor, the corresponding `try` and `catch` blocks must also be in the scope of the destructor or the results cannot be predicted.

Dynamic Link Library (DLL) Support

To use DLLs, you must use one of the following compilers:

- IBM C/C++ for MVS/ESA Version 3 Release 2
- IBM OS/390 C/C++ Version 1 Release 2 or later.

Note: There is no IBM VM compiler that supports the DLL compiler option. All applications that use the DLL compiler option must be compiled on an IBM MVS system. See the *TPF Migration Guide: Program Update Tapes* for more information about C/C++ compilers.

Several terms are defined to help you understand DLLs. Also see "E-Type Program Chart and Program Attributes" on page 56 for more information about where DLLs and DLL applications fit into the realm of TPF executable E-type programs and their characteristics and attributes.

See the IBM C/C++ programming guide and user's guide for the compiler used by your installation for a more thorough discussion of DLL support. See "IBM High-Level Language Books" on page xix for a list of C and C++ compiler publications on the System/390 platform.

Terminology and Concepts

Definition side-deck	A directive file that contains an `IMPORT` control statement for each function and variable exported by the DLL. When you build a DLL, a definition side-deck is automatically created and written to the `SYSDEFSD DDname` by the prelinker. You must include this definition side-deck when you prelink a DLL application that imports any of those functions or variables from a DLL.
Dynamic link library (DLL)	A collection of one or more functions or variables gathered in a load module and executable or accessible from a separate DLL application load module.
DLL application	An application that can reference imported functions or imported variables in a DLL.
Function descriptor	An internal control block that contains the function address and its associated writable static area (WSA). In the TPF system, a function descriptor can be thought of as a dynamic linkage call stub in contrast to the static linkage call stubs that are generated offline by the DLM stub generator tool (STUB) and the library interface tool (LIBI) before link-edit time.
Variable descriptor	An internal control block that contains the variable address. This control block is a dynamic linkage call stub.

The following are concepts to be aware of:

- Imported functions and variables are those that are not defined in the load module where a reference to them is made but are defined in a referenced DLL.

- Nonimported functions are those that do not use DLL linkage. Nonimported variables are those that are defined in the same load module where a reference to them is made.
- Exported functions and variables are defined in one load module and can be referenced from another load module.

Linkage

The connection or link between the DLL application that uses the DLL and the DLL functions or variables is made dynamically while the application is being run rather than statically when the application is built. You can, therefore, call a function or use a variable in a load module other than the one that contains the definition. You can use DLLs both implicitly and explicitly. When an application calls an imported function or references an imported variable, the DLL is implicitly loaded. This is referred to as *load-on-call*. For an explicit call, the application uses explicit source-level calls to one or more run-time services to connect the reference to the definition. These connections are made at run time.

The CISO run-time library has been updated with several application programming interfaces (APIs) that allow a load module to explicitly call these run-time services:

- `dllload`, which loads the DLL and connects it to the application
- `dllqueryfn`, which obtains a pointer to a DLL function
- `dllqueryvar`, which obtains a pointer to a DLL variable
- `dllfree`, which frees a DLL loaded with `dllload`.

See the *TPF C/C++ Language Support User's Guide* for more information about these services.

Functions or variables in DLLs can be called or referenced only through DLL linkage. The DLL cannot be called with TPF enter/back services.

Definition Side-Deck

When you link-edit a DLL or DLL application, in addition to specifying the included object files, you must also specify the *definition side-deck* inputs if the DLL or DLL application imports from other DLLs. A definition side-deck is automatically generated by the prelinker and contains `IMPORT` control statements for functions and variables exported by a DLL. The linkage editor resolves external functions in the following search order:

1. Those functions explicitly included in text decks (this is from the original list of included object files in a load module build script)
2. Those functions in included `EXPORT` data set members
3. Those functions for which static stubs exist in stub data sets concatenated under the `SYSLIB` data set. These stubs are created with the STUB tool.

Using sample load module FOOG, the following is an example of code that exports functions and variables and its resulting definition side-deck:

```
#pragma export(foo)
#pragma export(goo)
int foo()
{
...
}
int goo()
{
...
}
```

```
int keep_it_hidden()
{
...
}
...

#pragma export (hooVar)
#pragma export (gooVar)
int hooVar;
int gooVar;
int keep_it_hidden_variable;

Definition Side-Deck produced:

      IMPORT CODE 'FOOG' foo
      IMPORT CODE 'FOOG' goo
      IMPORT DATA 'FOOG' hooVar
      IMPORT DATA 'FOOG' gooVar
```

The C load module build tool (CBLD) has been updated to handle definition side-decks and DLLs. See "ISO-C Load Module Build Tool (CBLD)" on page 329 for more information about CBLD.

DLLs and Subsystem Dependencies

User exit function `is_dll_user_ss_shared` is in source file USUD, which is in DLM USUD. Adding shared user subsystem DLL names to the table in this module allows for those DLLs to be accessed by applications running in other subsystems. See *TPF System Installation Support Reference* for more information about this user exit.

Summary

You have been introduced, on a very high level, to several types of terminology:

- C language (as in functions, locales, header files, and C++ language)
- DLLs (as in DLLs, DLL applications, definition side-decks, and exporting)
- TPF (as in ECB and globals)
- Architectural (the AOLA, LIBVECs, stack, and the TPF API).

We have covered the basic concepts of TPF support for C/C++ application programming. It is time to move on to different parts of this publication, depending on whether you are interested in installing TPF C/C++ language support or writing application programs. If you are an application programmer, continue to "Writing TPF Application Programs in C and C++" on page 133. If you are a system programmer, skip to "Customizing C/C++ Language Support" on page 301.

Understanding TPF MQSeries Support

This chapter provides an overall understanding of TPF MQSeries support. Cross references are included throughout this information that direct you to more detailed explanations of certain functions or entities.

TPF MQSeries Client Support

TPF MQSeries client support provides a Message Queue Interface (MQI) client on the TPF system to enable the complete MQSeries application programming interface (API). MQSeries clients use MQI channels to communicate with remote MQSeries servers.

MQSeries Client

When an application connects to a queue manager other than the local queue manager, a connection is established with a remote MQSeries server. Each MQSeries API function (for example, MQPUT, MQGET, and so on) is then delivered to a remote MQSeries server over LU 6.2 or Transmission Control Protocol/Internet Protocol (TCP/IP) communications links and then processed by the MQSeries server. The queues themselves reside on the server, not on the TPF system. The API available to the application is determined by the API available on the remote server. For a complete description of the MQI, see *MQSeries Message Queue Interface Technical Reference*. For more information about MQI clients, see *MQSeries Clients* and the *MQSeries Distributed Queue Management Guide*.

MQI Channel Directory

An MQSeries client communicates with an MQSeries server by using an MQI channel, which is used to transfer MQI call requests from the client to the server, and responses from the server back to the client.

MQI channels differ from message channels (that are used to connect queue managers) in two ways:

- An MQI channel is bidirectional. One MQI channel can be used to send requests in one direction and responses in the opposite direction.

  With message channels, data can be passed in one direction only. If two-way communication is required between two queue managers (for example, when reply messages are to be sent to the same queue manager that handled an initial request message), two message channels then are required:
  - One message channel to handle messages that move in one direction
  - The other message channel for messages that move in the opposite direction.
- Communication on an MQI channel is synchronous. When an MQI request is transmitted from a client to a server, the MQSeries client product must wait for a response from the server before it can send the next MQI request.

  With message channels, the message traffic on the channel is time-independent. Multiple messages can be sent from one queue manager to the other without the sending queue manager having to wait for any replies from the receiving queue manager.

The TPF system maintains a maximum of 50 channel definitions in the MQI channel directory. The ZMQID ALTER, ZMQID DEFINE, ZMQID DELETE, and ZMQID

DISPLAY commands are used to maintain the MQI channel directory. See *TPF Operations* for more information about the ZMQID ALTER, ZMQID DELETE, and ZMQID DISPLAY commands.

TPF MQSeries Local Queue Manager Support

When an application connects to the queue manager, the application can connect to the local queue manager running on the TPF system by specifying the name of the local queue manager in the MQCONN function. All subsequent MQI function requests will be serviced by the local TPF queue manager. The MQI functions that are available to the local queue manager are more restrictive than most remote servers. See the *TPF C/C++ Language Support User's Guide* for more information about the MQI functions that are available.

Supported Queue Types

The TPF local queue manager supports the following queue types:

- Alias
- Local
- Remote.

These queue types are defined using the ZMQSC DEFINE QA, ZMQSC DEF QL, or ZMQSC DEF QR commands. The following are the two types of local queues:

- Normal
- Transmission.

Normal local queues physically reside in the TPF system and messages on local queues are retrieved by applications using the `MQGET` function. Applications can add messages to local queues for processing by a TPF application by using the `MQPUT` function.

Transmission queues contain messages that are destined for a remote system. In fact, the transmission queue is also physically located in the TPF system, but applications do not normally get and put messages directly to them. When an application puts a message to a remote queue, the TPF queue manager, in turn, determines on which transmission queue to put the message. At some point, the channel associated with that transmission queue takes the messages from that queue and sends them to the remote system.

With alias queues, the system administrator can define an alias queue that is opened by an application. However, unknown to the application, the queue that is actually opened is the target of the alias queue, which is some other local queue or local definition of a remote queue. In this way, the administrator manages the queues that are processed by applications. The application code never has to change to satisfy changes in queue names.

Starting TPF MQSeries Applications Using Triggers

TPF MQSeries provides a facility that allows you to automatically start an application when messages arrive on a queue. This facility is known as triggering. The ZMQSC ALT QL and ZMQSC DEF QL commands support the following trigger types:

First Trigger first processing occurs the first time a message arrives on a queue by setting a trigger when an application attempts to read (`MQGET`) a message from an empty queue. The next message that arrives on the queue, triggers a program in the process object associated with the queue. If no process object is associated with the queue, the TPF MQSeries queue trigger user exit ,CUIR, is called.

Every Trigger every processing occurs every time a message arrives on a queue. Every time a message arrives on the queue, the TPF system creates a new ECB and triggers a program in the process object associated with the queue.

When the TPF MQSeries queue trigger user exit, CUIR, is called, the TPF system passes the message queuing message descriptor (MQMD) and message queuing trigger message (MQTM) structures on data level 0 of the entry control block (ECB) to CUIR. CUIR can interpret this data and pass control to the appropriate application for processing the message. The MQGET, MQPUT, and MQPUT1 C functions define the values that are passed in the MQMD structure. The ZMQSC ALT PROC, ZMQSC DEF PROC, ZMQSC ALT QL, and ZMQSC DEF QL commands define the values that are passed in the MQTM structure. The MQMD structure is as follows:

```
typedef struct tagMQMD {
    MQCHAR4   StrucId;           /* Structure identifier */
    MQLONG    Version;           /* Structure version number */
    MQLONG    Report;            /* Report options */
    MQLONG    MsgType;           /* Message type */
    MQLONG    Expiry;            /* Expiry time */
    MQLONG    Feedback;          /* Feedback or reason code */
    MQLONG    Encoding;          /* Data encoding */
    MQLONG    CodedCharSetId;    /* Coded character set identifier */
    MQCHAR8   Format;            /* Format name */
    MQLONG    Priority;          /* Message priority */
    MQLONG    Persistence;       /* Message persistence */
    MQBYTE24  MsgId;             /* Message identifier */
    MQBYTE24  CorrelId;          /* Correlation identifier */
    MQLONG    BackoutCount;      /* Backout counter */
    MQCHAR48  ReplyToQ;          /* Name of reply-to queue */
    MQCHAR48  ReplyToQMgr;       /* Name of reply queue manager */
    MQCHAR12  UserIdentifier;    /* User identifier */
    MQBYTE32  AccountingToken;   /* Accounting token */
    MQCHAR32  ApplIdentityData;  /* Application data relating to
                                    identity */
    MQLONG    PutApplType;       /* Type of application that put the
                                    message */
    MQCHAR28  PutApplName;       /* Name of application that put the
                                    message */
    MQCHAR8   PutDate;           /* Date when message was put */
    MQCHAR8   PutTime;           /* Time when message was put */
    MQCHAR4   ApplOriginData;    /* Application data relating to origin */
    MQBYTE24  GroupId;           /* Group identifier */
    MQLONG    MsgSeqNumber;      /* Sequence number of logical message
                                    within group */
    MQLONG    Offset;            /* Offset of data in physical message
                                    from start of logical message */
    MQLONG    MsgFlags;          /* Message flags */
    MQLONG    OriginalLength;    /* Length of original message */
   } MQMD;
```

The MQTM structure is as follows:

```
typedef struct tagMQTM {
    MQCHAR4    StrucId;      /* Structure identifier */
    MQLONG     Version;      /* Structure version number */
    MQCHAR48   QName;        /* Name of triggered queue */
    MQCHAR48   ProcessName;  /* Name of process object */
    MQCHAR64   TriggerData;  /* Trigger data */
    MQLONG     ApplType;     /* Application type */
```

```
   MQCHAR256  ApplId;       /* Application identifier */
   MQCHAR128  EnvData;      /* Environment data */
   MQCHAR128  UserData;     /* User data */
  } MQTM;
```

When a process is called, the TPF system passes the MQTMC2 structure to the process. The ZMQSC ALT PROC, ZMQSC DEF PROC, ZMQSC ALT QL, and ZMQSC DEF QL commands define the values that are passed in the MQTMC2 structure. The MQTMC2 structure is as follows:

```
typedef struct tagMQTMC2 {
  MQCHAR4    StrucId;      /* Structure identifier */
  MQCHAR4    Version;      /* Structure version number */
  MQCHAR48   QName;        /* Name of triggered queue */
  MQCHAR48   ProcessName;  /* Name of process object */
  MQCHAR64   TriggerData;  /* Trigger data */
  MQCHAR4    ApplType;     /* Application type */
  MQCHAR256  ApplId;       /* Application identifier */
  MQCHAR128  EnvData;      /* Environment data */
  MQCHAR128  UserData;     /* User data */
  MQCHAR48   QMgrName;     /* Queue manager name */
} MQTMC2;
```

See *TPF C/C++ Language Support User's Guide* for more information about the MQGET, MQPUT, and MQPUT1 C functions. See *TPF Operations* for more information about the ZMQSC ALT PROC, ZMQSC DEF PROC, ZMQSC ALT QL, and ZMQSC DEF QL commands. See *TPF System Installation Support Reference* for more information about the TPF MQSeries queue trigger user exit.

Message Routing

The TPF system supports the following methods when resolving queue names.

Local Definition of Remote Queues

To remove the burden of having the application determine the queue manager and queue to receive its message, the MQSeries administrator can define a local definition of a remote queue that specifies the actual destination queue manager and destination queue name. The application opens a local name for the queue, and the TPF queue manager will then substitute the specified queue manager and queue name and put the message on the specified transmission queue. For more information about defining a local definition of a remote queue, see the ZMQSC DEF QR command in *TPF Operations*.

Queue Manager Aliasing

The TPF system also supports queue manager aliasing. Here, the name of the remote queue is known, but not the name of the remote queue manager. When the application opens a queue specifying a queue manager, the TPF system will look up the name of the queue manager and substitute the queue manager that is specified. Queue manager aliasing is accomplished by leaving the RNAME field blank in the ZMQSC DEF QR command.

Queue Manager Name as Transmission Queue Name

In addition to a local definition of remote queues and queue manager aliasing, the system administrator can send a message to an adjacent queue manager if the name of the queue manager that is opened by the application is the same as a transmission queue.

Middle Hop Routing

Messages that are received by TPF MQSeries local queue manager channels may not be destined for the TPF queue manager. The TPF receiver channel calls the local TPF queue manager to resolve the name of the destination queue manager and queue name for each message it receives. The queue manager and queue name are resolved according to the rules previously stated, and the message is put on the appropriate transmission queue.

Processor Unique Queues versus Processor Shared Queues

Turbo enhancements for TPF support of MQSeries local queue manager provides a performance enhancement that makes processor unique queues (which are defined by specifying NO for the COMMON parameter on the ZMQSC DEF QL command) memory resident. Processor shared queues (which are defined by specifying YES for the COMMON parameter on the ZMQSC DEF QL command) reside in TPF collection support (TPFCS). Before this enhancement, all queues resided in TPFCS. Now, processor unique queues reside in memory and use checkpoint records and the recovery log as a repository for persistent data (such as the messages). Only local normal queues can be defined as processor shared. See *TPF Operations* for more information about the ZMQSC DEF QL command. See *TPF Database Reference* for more information about recovery logs.

Monitoring Queue Depth

The TPF system provides a queue depth monitor for processor unique queues. When the queue depth on a transmission queue exceeds the value specified by the administrator on the QDEPTHHI parameter using the ZMQSC DEF QL command, a warning message is sent to the operator console. This could mean that the queue is stalled and may need operator intervention. The warning message is sent to the console every *xx* seconds until the queue goes below the QDEPTHHI value (where *xx* is the interval that is determined by the administrator via the QDT parameter on the ZMQSC DEF MQP command). See *TPF Operations* for more information about the ZMQSC DEF QL command.

Channels

The TPF MQSeries local queue manager supports two channel types that connect to remote MQSeries systems:

- A sender channel, which must connect only to a remote receiver channel
- A receiver channel, which accepts one connection request at a time, only from remote sender channels.

Receiver channels make use of the TPF Internet daemon. When a remote sender channel first connects to the TPF system over Transmission Control Protocol/Internet Protocol (TCP/IP), it sends a connection request to port 1414, which is the standard MQSeries port. System administrators must set up an Internet daemon listener on that port that, once the connection request is received, passes control to the TPF MQSeries receiver channel session initiation program (CMQL). To set up the Internet daemon listener for MQSeries, you must add an MQSeries server by entering the following.

```
ZINET ADD S-MQS P-TCP MODEL-AOR PORT-1414 PGM-CMQL AORL-8
```

This is required before establishing a connection between remote sender channels and TPF receiver channels. It is possible to change the TCP/IP port that is used for these connections. If you establish an Internet daemon listener on a different port for the MQSeries server, you need to specify the same port in the connection name when defining the sender channel on the remote MQSeries system.

Two channel *speeds* are supported for both sender and receiver channels: normal and fast.

When sending messages over normal speed sender channels, persistent and nonpersistent messages are included in batches and receipt confirmation is required before the messages are deleted from the transmission queue.

Persistent and nonpersistent messages can be sent over fast sender channels. When sending messages over fast sender channels, only persistent messages are included in batches. Nonpersistent messages are sent outside of the batch and are deleted from the transmission queue without receiving receipt confirmation. Nonpersistent messages are never sent again during channel recovery procedures.

When receiving both persistent and nonpersistent messages over normal receiver channels, the messages are processed as part of a batch, and receipt confirmation is sent for the entire batch of messages. Once the confirmation is sent, the messages appear on local TPF MQSeries queues or are put on transmission queues destined for another queue manager if the TPF queue manager is not the target queue manager. Applications must retrieve messages from the local queues by using the `MQGET` function.

When receiving nonpersistent messages over a fast receiver channel, the message is not filed and is assumed to be destined for a traditional non-MQSeries application. To obtain significant performance throughput for these messages, they are given directly to TPF applications by using the TPF-unique MQSeries ROUTC Bridge function. The messages never appear on a queue. Persistent messages received over fast receiver channels are processed as if the message was received over a normal receiver channel.

MQSeries ROUTC Bridge

TPF local queue manager support includes a TPF-unique mechanism for passing nonpersistent messages received over fast channels directly to traditional TPF applications. In this way, TPF customers can take advantage of MQSeries-oriented networks for delivering *older* traditional, high-speed TPF-type messages to TPF applications. The MQSeries message is converted into TPF AM0SG format and given to TPF message router program COA4 for routing the message to an application. A user exit is provided that gives you the opportunity to assign a line number, interchange address and terminal address (LNIATA) to the message before giving it to the application. In addition, the terminal address table (WGTA) for that LNIATA is marked with an MQSeries indicator, so when the application responds to the message using the ROUTC bridge, the message is intercepted and converted back to an MQSeries message format. See *TPF System Installation Support Reference* for more information about user exits.

Transmission Queues: Swinging

The TPF local queue manager provides a unique feature that redirects messages originally destined for a transmission queue to an alternate transmission queue. If, for example, a channel is stalled or the remote receiver channel is down, messages can be moved to a transmission queue that has an active channel. All messages that were previously on the original transmission queue are moved to the new transmission queue, and all new messages put to the original transmission queue are actually added to the new transmission queue. The ZMQSC SWQ command is used to perform this function. See *TPF Operations* for more information about the ZMQSC SWQ command.

Transaction Manager

With the release of turbo enhancements for TPF support of MQSeries local queue manager, the TPF transaction manager was enabled to control MQSeries API functions. This means that MQSeries `MQPUT`, `MQGET`, and `MQPUT1` API functions will participate in transaction scopes. Before this, an `MQPUT` function in a commit scope resulted in the message being immediately put on the queue and the queue was locked until a `tx_commit` function or `tx_rollback` function was issued. The transaction manager had no knowledge of the MQSeries APIs. With turbo enhancements for TPF support of MQSeries local queue manager enhancements, `MQPUT` and `MQGET` functions become visible to other processes during `tx_commit` function processing and the queue is only locked during `tx_commit` function processing. With the transaction scopes in place for MQSeries APIs, the behavior of these APIs will change for those applications that already have transaction scopes surrounding the MQSeries APIs.

Browsing TPF MQSeries Processor Shared Queues

Because the TPF MQSeries implementation for processor shared queues uses TPF collection support (TPFCS) for its database, there are several named collections that you can browse by using the TPFCS browser. To display the named collections, enter the ZBROW NAME command with the DISPLAY and ALL parameters specified. Ensure you set the browser qualifier to MQSeries data store MQSC*xxx*, where *xxx* is the subsystem name (for example, MQSCBSS). See *TPF Operations* for more information about the ZBROW QUALIFY command used to set the browser qualifier, and the ZBROW NAME command.

Trace

Communications trace and function trace are included in TPF MQSeries support.

With communications trace, the data sent and received over channels is included in a trace block attached to the channel definition. Much of the data will not be formatted because the contents of the message flows are considered proprietary and confidential.

With function trace, you can trace function calls in the three different areas of MQSeries:

- Administrative (ZMQSC commands)
- Queue manager
- Communications.

Each function that is called and returned creates an entry in a trace block that is attached to the ECB. You can use trace data for problem determination. You can use function trace to trace individual channels or queues, or all channels or all queues.

Trace data can be sent to either the console, the RTA tape, or both the RTA tape and console. Only send trace data to the console in a test system environment so the console does not become flooded with trace messages. See *TPF Program Development Support Reference* for more information about trace.

Administering Your Local Queue Manager

The administrative functions in the TPF implementation of an MQSeries local queue manager are similar to, but not exactly the same as, other platforms. Some of the functions are unique because of the loosely coupled nature of TPF systems, while other differences result because TPF maintains some of the MQSeries object definitions in system heap for performance.

Defining the MQSeries Profile

When you define the MQSeries profile, the TPF system automatically provides a system queue called DEAD.LETTER.QUEUE and a SPECIAL.RECOVERY.QUEUE; these are the only default queues provided. Other platforms provide several other default queues. Any message that arrives at the TPF system whose destination queue name cannot be resolved is put on the dead-letter queue. All users are expected to create a monitor for the dead-letter queue that determines what to do with messages that arrive there. The special-recovery queue is used by transaction services to recover uncommitted messages on processor shared queues.

Defining Processor Shared Queues

In a loosely coupled environment, normal local queues can be shared between processors or they can be processor unique. When you define a local queue by using the ZMQSC DEF QL command, you can specify that you want all processors to see the queue by specifying a value of YES for the COMMON parameter. When you specify YES for the COMMON parameter, a single TPF collection support (TPFCS) persistent identifier (PID) is used for the queue that all of the processors can see. Messages added to the queue from one processor can be retrieved from another processor. If you specify a value of NO for the COMMON parameter, the queue resides in memory for each processor and is made persistent using the TPF recovery log.

Once you have specified whether the queue is shared or not, you cannot change this attribute using the ZMQSC ALT QL command. To change it, you will need to delete the queue and redefine it with the new attribute.

All transmission queues are processor unique because they all are associated with channels that are also, by definition, processor unique. Remote queues are really virtual and actually get resolved to physical transmission queues.

CPU Parameter for Channel Definitions

Channels, by nature, are processor unique. Each processor establishes an MQSeries channel connection to an adjacent MQSeries system. As a matter of convenience, the TPF system provides a CPU parameter for the ZMQSC DEF CHL and ZMQSC ALT CHL commands. Therefore, if you are on processor A, you can define or change channels on processor B even while processor B is down.

Altering Channels

Because channel definitions reside in memory and you do not want to lock the tables for each access, changes to these definitions do not occur immediately. For changes to channel definitions, the channel must be stopped and restarted.

Checkpoint

All the memory queues and some channel data is filed to fixed file records (#IMQCK) on a regular basis. This is called MQSeries checkpointing. The file copy of the queues and channels, together with the data written to the recovery log, are how The TPF system ensures that all data is made persistent during an IPL of a TPF system. TPF restart will rebuild the queues and channels to exactly the same state as before the IPL by using the fixed file checkpoint with the data found on the recovery log.

The size of the recovery log must be enough to accommodate all logging activity for all TPF resource managers, (not just the TPF MQSeries resource manager) between each successful completed checkpoint. The checkpoint interval is 5 seconds.

Ensure that you have enough fixed file checkpoint records (#IMQCK) to fit all the queues and channel data. See *TPF System Generation* for information on how to determine the correct number of records to allocate for checkpointing.

Sweep

Because processor-unique queues reside in memory, TPF system defaults are set to move queues from system work block (SWB) memory to TPF collection support (TPFCS) records if the queue is not processed at a reasonable rate. The processing rate is considered unreasonable when the number of messages on the front and rear message list of the local memory queue is greater than half the number of outstanding GET requests.

Tuning Memory Allocation

To accommodate processor unique queues that reside in memory, you will need to allocate significantly more memory resources to system work blocks (SWBs). To calculate the number of SWBs required to accommodate message traffic, assume each SWB can hold 1000 bytes of message data plus an additional 500 bytes of message header for each message. Therefore, if the average message size is 750–850 bytes of message data, each message will fit in one SWB. If the message size is 4096 (on the average), each message will then require five SWBs.

Deleting Queues

A local queue will be marked as delete pending until it is empty on every active processor in the complex. Enter the ZMQSC DEL QL command with the purge parameter specified to bypass this requirement for specified processors and delete all messages on the queue for this processor. Messages can be removed from a queue marked delete pending, but no messages can be added to them. An attempt to issue an `MQPUT` function to a queue that is pending a deletion results in an MQRC_Q_DELETED reason code. See *TPF Operations* for more information about the ZMQSC DEL QL command.

Understanding TPF Transaction Services Concepts

TPF transaction services support is designed to help application programmers by ensuring a consistent view of the database. A consistent view of the database helps in the following ways:

- Reducing application complexity by having to write and maintain fewer error recovery routines
- Reducing application development cycle time because you know you do not have to worry about a partially-updated database. Either ***all*** of your file changes have completed or ***none*** of them have. You can thereby reduce the amount of error recovery code you have to write.
- Increasing programmer productivity because the combination of reduced application complexity and development cycle time means that you can add more function or you can get your application to market that much sooner.
- Increasing application reliability because the order of updates is no longer of paramount importance. At any time, the application can stop processing and request that the TPF 4.1 system ignore all of its previous updates.

This overview of TPF transaction services includes discussions about defining the commit scope, requesting that data be written to the DASD surface, and understanding how to operate inside and outside the commit scope.

The term *commit scope* is used throughout this chapter to refer to a unit of work that groups together a set of database updates. These updates can then be written, or *hardened*, to the DASD surface as a group at the same time or rejected as a group (where no hardening takes place).

The application view of the commit scope is through the macro interface or application programming interface (API). Therefore, it is necessary to understand the macro API and how it is affected by the commit scope.

Defining a Commit Scope

You need to explicitly define the start and end of a commit scope in the application program; this is commonly referred to as a *begin transaction*. The TPF system provides a subset of TX functions (defined by the X/Open TX interface) to the application to begin and end (that is, commit or roll back) a transaction:

Table 10. TPF Transaction Services Begin and End Transactions

C Function	Assembler Macro
`tx_begin`	TXBGC
`tx_commit`	TXCMC
`tx_rollback`	TXRBC
Note: In this publication, these functions and macros are referred to as begin, commit, and rollback transactions, respectively.	

Additionally, the TPF system provides the following extension (to X/Open) functions to the application to suspend or resume a transaction:

Table 11. TPF Transaction Services Suspend and Resume Transactions

C Function	Assembler Macro
`tx_suspend_tpf`	TXSPC

Table 11. TPF Transaction Services Suspend and Resume Transactions (continued)

C Function	Assembler Macro
`tx_resume_tpf`	TXRSC
Note: In this publication, these functions and macros are referred to as suspend and resume transactions, respectively.	

Commit Scope Nesting

Commit scope nesting provides a powerful mechanism for fine-tuning the scope of a rollback transaction (that is, the `tx_rollback` C function or the TXRBC assembler macro) in applications with a complex structure. The term *nesting* is used whenever a begin transaction is requested and a commit scope is already active. An application may consist of multiple processes where each process is responsible for its own commit scope. This environment can be represented by *root* and nested commit scopes.

Root Scope

The root commit scope is the first commit scope that is activated by the application. Figure 12 shows a begin and a commit transaction in a simple commit scope. The begin transaction could be either the TXBGC macro or the `tx_begin` C function. The commit transaction could be either the TXCMC macro or the `tx_commit` C function.

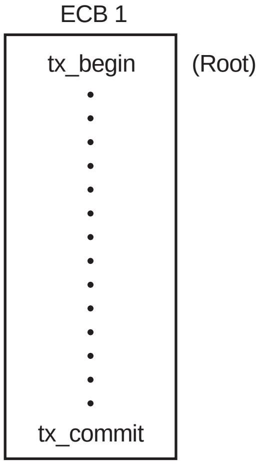


Figure 12. Root Commit Scope

Nested Scope

A nested commit scope is a commit scope that is activated after the root scope has been activated. A root commit scope may have many nested scopes. A nested scope may, in turn, have its own nested scopes. Figure 13 on page 91 shows a root scope, which is started with the first begin transaction followed by three nested scopes. The first begin transaction is the highest-level scope. The fourth begin transaction is the lowest-level scope; this is the last commit scope to open before the root scope is either committed or rolled back.

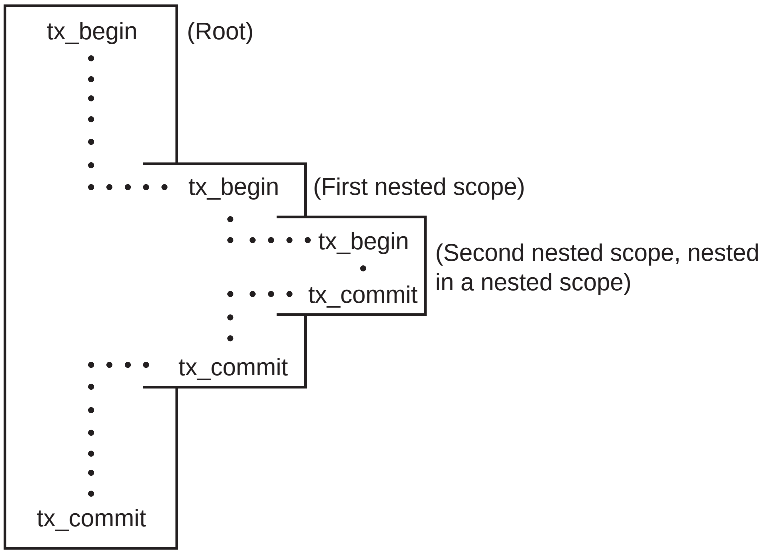


Figure 13. Root Commit Scope with Nested Scopes

Suspending a Commit Scope

Suspending the current commit scope (the root and all nested levels) permits changes to be made to DASD records outside of the current commit scope set, including any nested commit scopes. You can define another commit scope to coordinate these out-of-scope changes. References to records in the suspended commit scope set are not permitted; they cause a transfer of control to the system error routine.

In the Commit Scope

The following explains how commit scope processing affects DASD requests, pool file addresses, and TPF MQSeries support.

DASD

In the commit scope, all DASD requests are satisfied first from the current commit scope set and, then up the chain of nested scopes (from lowest to highest), and finally from outside the commit scope.

Finding Records

The TPF system first searches in the commit scope set for the record; if it is not found, normal DASD retrieval takes place from VFA or the DASD surface. Once it is found, the record is attached to the ECB. Find requests are not added to the commit scope.

Filing Records

The TPF system writes the record to the commit scope buffer; writing (or hardening) the data to the DASD surface occurs only when you enter the commit transaction. File requests are always added to the commit scope.

The action of the FILEC macro depends on the settings returned from the record ID attribute table (RIAT) and the RIAT user exit. These settings include the control of logging, record cache options, and VFA options. The FILEC actions that are controlled by the RIAT indicators are set at FILEC time, but do not take place until the record has been committed.

Holding Record Locks

Record lock holding occurs at two levels: the ECB level and the commit level. Following a find-and-hold type macro, record locks are held at the ECB level. If an ECB releases a lock that is held at the ECB level using an unhold-type macro while in a commit scope, the lock is no longer held at the ECB level; it is then held at the commit level.

A record lock that is held at the commit level becomes held at the ECB level if the ECB in the commit scope, or a commit scope nested from the commit scope, issues a find-and-hold type macro.

A record lock that is held at the commit level becomes unheld when the root commit scope commits.

A record lock held at the ECB level is released if an ECB issues an unhold-type macro outside of a commit scope. Table 12 on page 93 is a matrix that shows what happens when locks are held and released and the effect if they are held inside or outside the commit scope. See *Unhold in commit scope* in column 1 for an example of how to read the matrix.

Operation
The action taken is that an unhold-type assembler macro or C function is entered in a commit scope.

Columns 2 and 3 represent current lock processing for the TPF system. This functions the same regardless of whether TPF transaction services processing is active.

Record Lock Is Not Held
If the record lock is not held, the TPF system dumps.

Record Lock Is Held in ECB
The lock is now held at the commit scope level.

Columns 4, 5, and 6 represent processing in the TPF system.

Record Lock Is Held in Current Commit Scope
If the lock is held in the current commit scope and an unhold-type macro or function is entered, the TPF system dumps.

Record Lock Is Held in Nested Commit Scope
The result is the same as for column 4 in that the TPF system dumps.

Record Lock Is Held by the ECB and the Hold Was Done in a Commit Scope
If the lock is held by the ECB and the hold was entered in a current commit scope, the lock is held at the commit scope level.

Table 12. Matrix of the Locking Scheme for DASD Inside and Outside a Commit Scope

Operation	**Current Lock Processing**		**Processing in a Commit Scope**		
	Record Lock Is Not Held	**Record Lock Is Held in ECB**	**Record Lock Is Held in Current Commit Scope**	**Record Lock Is Held in Nested Commit Scope**	**Record Lock Is Held by the ECB and the Hold Was Done in a Commit Scope**
Commit	Not held	Held ECB	Release lock	Held at commit level	Held at ECB level
Rollback	Not held	Released if acquired in the scope; otherwise, held at ECB level	Release lock	Held commit	Held at commit level if acquired in the scope; otherwise, held at ECB level
Unhold in commit scope	Dump	Held at commit level	Dump	Dump	Held at commit level
Unhold suspended commit scope	Dump	Lock is released	Dump	Dump	Dump
Unhold other ECB	Dump	Dump	Dump	Dump	Dump
Hold in commit scope	Held at ECB level	Dump	Held at ECB level	Held at nested commit and ECB levels	Dump
Hold suspended commit scope	Held at ECB level	Dump	Dump	Dump	Dump
Hold other ECB	Held at ECB level	Queued	Queued	Queued	Queued

WAITC Processing

WAITC will normally suspend the ECB until the requested I/O has completed. Most WAITC processing happens normally in a commit scope. One exception is FILNC and WAITC. FILNC-detected DASD surface errors that are currently reported on WAITC cannot be reported if the FILNC operation is in a commit scope because the actual filing of the record does not take place until you enter a commit transaction. Hardware errors, normally seen at WAITC completion, will not be returned.

Pool File Addresses

TPF transaction services processing does not change the retrieval process for pool file addresses; that is, there is no change when an ECB gets a pool address from the TPF system. Nothing different happens for these addresses. When you enter a commit transaction, the retrieval process is still the same. However, these addresses will be released if you roll back the transaction.

The way release pool address processing works changes in a commit scope; the timing of the release of pool addresses is different. Because of the possibility of a rollback transaction, pool address release requests are held until the commit transaction is completed. This applies to both single releases and chain (RLCHA) releases.

TPF MQSeries Support

Updates to local queues, which are managed by the TPF local queue manager, participate in the commit scope. The MQPUT, MQPUT1 and MQGET functions are part of the transaction that can be committed or backed out by the application. Updates to queues that are managed by a remote MQSeries server (via TPF MQSeries client support) do not participate in the commit scope and are processed outside the unit of work.

When an application issues an MQPUT or MQPUT1 function in an open commit scope, the message appears on the queue in an uncommitted state. No other ECBs can access the message unless the application commits the unit of work. The same ECB can see the message on the queue if it issues an MQGET function even though the commit scope is still open.

When an application issues an MQGET function while in an open commit scope, the message is retrieved from the queue and given to the application, but is not deleted from the queue until the application commits the unit of work. No other ECBs can access the same message until the root commit scope is rolled back.

When an application issues an MQGET function to a processor shared queue while in an open commit scope, the message is retrieved and deleted from the queue and given to the application. The message is put back on the queue if the transaction ends abnormally or if the application rolls back the transaction.

No other MQSeries APIs participate in the commit scope. For instance, a `tx_rollback` function does not imply that a queue that was opened is automatically closed.

Ending a Commit Scope

A commit scope can be ended in the following ways:

- Entering a commit transaction

This is the normal way to end a commit scope and causes all changes to be either written to the DASD surface, the recovery log, or reflected up to the next higher level commit scope.

- Entering a rollback transaction

  This is the abnormal way to end the commit scope; all DASD changes are discarded with this method. The writing of changes to the DASD surface ends abnormally. Any release pool address requests that were made in the commit scope are discarded. Messages put to a queue through the MQPUT function are removed, and messages retrieved from a queue through the MQGET function are put back.
- Through exit or system error processing.

  If the ECB either exits or causes a system error in the commit scope, an implied root commit scope rollback transaction is entered. If the ECB has suspended root scopes, these are also rolled back.

Commit scopes are processed according to the following rules:

- Commit rules
  - When a commit transaction is entered in a nested commit scope, any file requests (functions or macros such as `filnc` or FILNC) that you entered in the nested commit scope are passed up to the next higher level commit scope. Essentially, when you enter a file request in a lower level (that is, nested) commit scope, followed by a commit transaction, the request becomes part of the higher level commit scope.

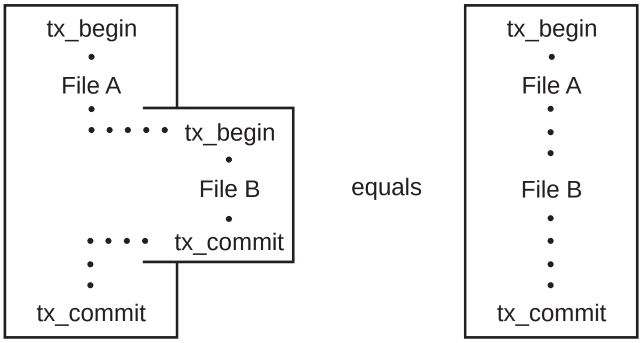


Figure 14. Example of a Nested Commit Scope

  - When a commit transaction is entered in a root commit scope, it causes all changes to be written to the DASD surface.

  When you enter a commit transaction on a commit scope, locks that are held at the commit scope level are either:
  - Copied into the next higher commit scope if the commit scope is a nested commit scope.
  - Released to the TPF system if the root commit scope is committed.

  When you enter a commit transaction on the current commit scope, all pool addresses acquired while in the commit scope are:
  - Made part of the next higher commit scope, if the commit scope is a nested commit scope
  - Given to the application without the possibility of automatic release by the TPF system, if the root commit scope is committed.

Note: If a system error exits the ECB before you enter a commit transaction, any get pool requests (that is, a `getfc` function or GETFC macro) are rolled back, which then releases the pool address back to the TPF system.

When you enter a commit transaction on the current commit scope, all requests entered in the commit scope to release file pool addresses are:

- Deferred until the root scope commits if the commit scope is a nested commit scope
- Returned to the TPF system if the root commit scope is committed.

When you enter a commit transaction on the current commit scope, all virtual file access (VFA) flush requests entered in the commit scope are:

- Deferred until the root scope commits if the commit scope is a nested commit scope
- Entered after all file-type macros are entered.

When entering a commit transaction on the current commit scope, all requests entered in the commit scope to change any queues managed by the local TPF MQSeries local queue manager (MQPUT, MQPUT1, or MQGET functions) are passed up to the next highest commit scope.

- When a commit transaction is entered in a root commit scope, it causes all messages put to all queues to become visible to all other ECBs. For processor shared queues, the message is visible to all other processors in the loosely coupled complex.
- When a commit transaction is entered in a root commit scope, it causes all messages retrieved from all queues to be permanently removed from the queue.

- Rollback Rule

  When you enter a rollback transaction, the following processing takes place:

  - All records that were written with file-type macros entered in the commit scope to online (non-general file) DASD are discarded. The records will have never been visible to any other ECBs.
  - All DASD locks that were acquired with find-and-hold type macros in the commit scope that is being rolled back will be released:
    - To a higher commit scope if they were held at the commit scope level by a higher level (nested) commit scope
    - To the TPF system if they were not previously held at the commit scope level.

    DASD locks that are obtained outside the commit scope are not affected by the rollback of a commit scope.

  All pool addresses acquired while in the commit scope are released back to the TPF system if the general file system (GFS) is active. If the GFS is not active, the pool addresses are lost until the next time you run the recoup utility. Pool addresses that are obtained outside the commit scope are not affected by the rollback transaction.

  - Entering a `getfc` function (or GETFC macro) to get a pool address within a commit scope, followed by the rollback of the scope, makes it seem as though the get request never took place.

  - Entering a `getfc` function (or GETFC macro) to get a pool address outside the current commit scope, followed by the rollback of the current scope, will have no effect on the get request. The pool address is still available for the application.

  All pool address releases that are entered while in the commit scope are ignored. The pool addresses, if acquired outside the commit scope, are still owned by the application.

  - Entering a get request (`getfc` function or GETFC macro) and a release request (`relfc` function or RELFC macro) of a pool address in a commit scope, followed by the rollback of the scope, makes it seem as though the requests never took place.
  - Entering a get request (`getfc` function or GETFC macro) and a release request (`relfc` function or RELFC macro) of a pool address outside the current commit scope, followed by the release of the address in the current scope and the subsequent rollback of the current scope, makes it seem as though only the get request and not the release request took place.
  - VFA flush requests that are entered in a commit scope are discarded if you enter a rollback transaction in the commit scope.

  All messages that were added to queues are removed from the queues and all system resources are released. All messages that were retrieved from the queues are restored to the queues.

- Visibility Rule

  All changes made by a nested scope become visible to the next higher level nested scope when you enter the commit transaction.

  When you enter the commit transaction on the root scope, all the changes that were made become visible to all ECBs.

  Changes made in a nested scope are not visible to higher level nested scopes, to other commit scopes, or to ECBs not running in a commit scope until you enter the commit transaction on the root commit scope.

Outside the Commit Scope

Because the starting of a commit scope is under ECB control, you can implement TPF transaction services support without affecting the way existing applications run. However, existing applications may be impacted by applications using this support.

Finding Records

Normal find processing is not affected by TPF transaction services support.

Find-with-hold processing may be affected; see “Holding Records” for more information.

Filing Records

Normal file processing is not affected by TPF transaction services support.

File-with-unhold processing may be affected, however; see “Holding Records” for more information.

Holding Records

Although there are no API changes for processes that are outside of the commit scope, you need to understand what effect the commit scope has on them. For requests outside of the commit scope, a commit-level hold is viewed the same as an ECB-level hold. This means that if the requested record is held at the ECB level or the commit level, the request will be queued. This processing could result in an increase in record hold duration and an increase in record hold deadlock conditions.

For example, look at the scenario in Figure 15. Program A, which is using TPF transaction services support, causes a deadlock to occur.

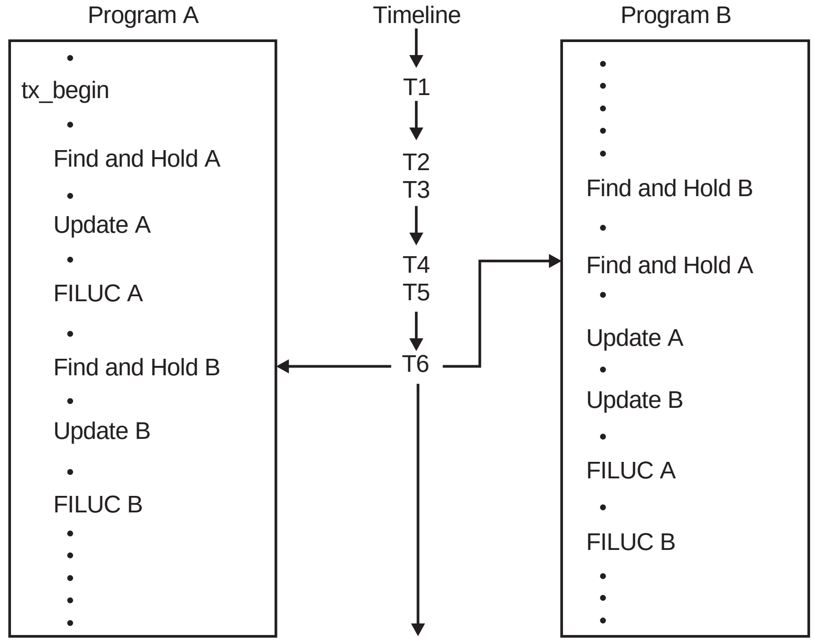


Figure 15. Example of a Deadlock Condition

T1 Program A starts a root commit scope by entering a begin transaction.

T2 Program A gets a hold lock on record A. The lock is added to the commit scope.

T3 Program B gets a hold lock on record B.

T4 Program B requests a hold on record A; record A is still held by program A so the request is queued and program B waits.

T5 Program A files and unholds record A; the lock for record A is still held by the commit scope. Program B will continue to wait.

T6 Program A requests a hold on record B. Record B is held by program B, which is still waiting, so the request is queued and program A waits, resulting in a deadlock condition.

Deadlock Detection

A deadlock detection routine is provided to assist with deadlock detection. This routine is a time initiated routine and is activated during restart or by the CRETC macro.

Deadlock detection processing is applied on loosely coupled systems and base-only systems. It consists of taking a snapshot of all of the record hold tables (RHTs) in the complex, merging them into a table, and then running the table through the deadlock detection algorithm.

To have a true event, a cutoff time (the earliest of the time stamps associated with each CPU snapshot of the RHT) determines whether an individual entry of an RHT of a CPU will be merged into the table. If a record is held within the cutoff time, the related RHT entry is merged into the table. Otherwise, the entry is discarded.

When the deadlock detection program is activated, the CPU that has the lowest ordinal number will be the master CPU. The master CPU merges all of the RHT snapshots that were sent by other CPUs and performs the deadlock detection routine.

If a deadlock is detected, a deadlock user exit (CLUD) is activated on the CPU where the deadlocked ECB is located. The deadlock detection routine takes an action as follows:

- If the return code from the user exit is 0, the ECB remains deadlocked.
- If the return code from the user exit is 4, the ECB will be scheduled to exit with dump D9.
- If the return code from the user exit is 8, the deadlock detection routine sets the CE1SUD and CE1SUG fields of the ECB to CJCSUHRD and CJCSUDLK (that is, X'81').

  The new deadlock detection bit in SUD and SUG is as follows:

  In MRLNQ, the bit is called:

  ```
  CJCSUDLK  EQU   X'01'           DEADLOCK DETECTION
  ```

  In EB0EB, the bit is called:

  ```
  CXSGDLK   EQU   X'01'           DEADLOCK DETECTION
  ```

  The waiting input/output block (IOB) associated with this ECB is removed from the waiting queue and the post-interrupt routine in the IOB is activated.

You can also use the ZECBL command with the E parameter to remove all the IOBs associated with the deadlocked ECB and force the ECB to exit with dump D9.

SYSTC switch SBDLOCK is defined for deadlock detection processing. You can turn it off using the ZSYSG ALTER command with the NODLOCK parameter.

Loosely Coupled and Multiple Database Function (MDBF) Considerations

In a loosely coupled complex, records that are held in LLF/CFLF control units while the transaction is active need to remain held during the recovery process. The TPF system will keep locks across a system IPL until the recovery log is processed to synchronize data record updates among all loosely coupled processors and ensure database integrity.

Exceptions

The following cases are exceptions to TPF transaction services processing.

- Unsupported functions
  - Processing of general files or general data sets is not considered part of the commit scope and is not affected by commit scope processing. All updates to general files are made at the time of the macro request using standard TPF 4.1 logic. A rollback transaction will not undo these updates.
  - Processing of find/file single and find/file special macros is not supported in the commit scope and causes the transfer of control to the system error routine.

- Functions that do not work

  Applications that do any of the following must ensure that data is committed; that is, the commit scope is ended before using any of the following:

  - Pass file address or file chains to other application functions or system functions that run either on a separate ECB or without an ECB
  - Use techniques that coordinate updates to both core and file records without relying on find-and-hold processing for concurrency control
  - Use *hold* techniques other than DASD holds.
- The following system functions will not work when called from a commit scope:
  - ROUTC, SENDC, and CVIx, where the ROUTC macro is issued. System error CEC136I is taken in this case. System error CEC137I is taken when the chain message is sent using CVIx.
  - TPF Database Facility (TPFDF).
  - Systems Network Architecture (SNA) message recovery.

Understanding TPF Collection Support

TPF collection support (TPFCS) is a service for managing the storage and retrieval of data on a TPF database. The data is stored in units known as *collections*. Collections are abstract representations of data having common attributes and functions. Persistent collections maintain their state after the entry control block (ECB) that creates them exits.

TPFCS can be considered an application development tool that integrates database functionality with the application. Potentially complicated data manipulation routines are not needed in application programs because their functionality is already included in the TPF 4.1 system. Furthermore, the TPF 4.1 system does not need to have any knowledge of the format of the data, so more control is given to the application and taken away from the TPF system.

This section discusses many of the application programming interfaces (APIs) used in writing TPFCS applications. For more information about these APIs, see the *TPF C/C++ Language Support User's Guide*.

Application Characteristics

This section provides information about some of the characteristics that are unique to TPFCS application programs.

TPFCS Environment Block

To access TPFCS data stores and collections, the application must do the following:

- Define the application to TPFCS
- Connect the application to the specified data store
- Create an environment block for your application.

Calling the `T02_createEnv` function accomplishes all of these goals.

The `T02_createEnv` function is issued once by each ECB for each data store in the database that is accessed by that ECB. In other words, if the application will be creating collections in multiple data stores, the application will have to issue a `T02_createEnv` for each data store. The environment block that is created is built in ECB private storage and, therefore, cannot be shared among ECBs and its pointer cannot be saved to be used by other ECBs.

A pointer to the environment block is returned by the `T02_createEnv` function so that your application can access the database. The environment block that is created is used on almost all the TPFCS function calls. The collection creation functions (`T02_create...`) use the environment block to determine in which data store to create the collection.

Because the TPFCS system can usually determine the correct data store to work with from the persistent identifier (PID) of the collection, the data store of the environment passed to a function call does not need to be the same as the data store in which the collection was created. For example, an environment could be created with the TPFDB data store and be used by an application to access any collection already created in any data store. However, if a target collection PID is not included on a function call (such as with the `T02_atDSdictKey` function), the appropriate data store environment must be used.

When the ECB has completed accessing the TPFCS database, enter a `TO2_deleteEnv` function call to allow TPFCS to clean up and release any allocated system resources that it is still holding.

Type Definitions

The following type definitions are found in the `c$to2.h` header file:

TO2_ENV_PTR	This type is defined as a pointer to void. A variable of this type is set by calling the `TO2_createEnv` function and passing a pointer to this pointer. It is set by TPFCS to point to the environment block, which is used on almost all TPFCS API calls.
TO2_PID	This type defines the TPFCS PID assigned to a collection when it is created and then used to reference to the collection until it is deleted from the TPFCS database.
TO2_PID_PTR	This type is defined as a TO2_PID pointer and should be set to point to a variable of type TO2_PID.
TO2_BUF_HDR	This type defines the returned TPFCS data buffer header returned on an element retrieval using such TPFCS functions as `TO2_at`, `TO2_atKey`, `TO2_atCursor`, and `TO2_key`. See "Returned Data Structures" on page 106 for the format of the buffer.
TO2_BUF_PTR	This type is defined as a TO2_BUF_HDR pointer.
TO2_ERR_CODE	This type is defined as a long integer that, on return from a `TO2_getErrorCode` function call, contains the actual error code value that is stored in the environment.
TO2_ERR_TEXT_PTR	This type is defined as a `char` pointer and is returned from a `TO2_getErrorText` call. It will point to text that describes the actual error that occurred.

Error Handling

When a TPFCS function is successful, it returns a positive value to the application program. When an error occurs in an attempt to process a TPFCS function, TPFCS sets an error code in the environment block and returns TO2_ERROR (which is defined as zero) to the application program. Once the application identifies that an error has occurred, it can query the environment block using the `TO2_getErrorCode` function to determine the specific error that has occurred. If additional text describing the error is desired, the error code can be passed to the `TO2_getErrorText` function. Error processing is handled the same way regardless of the type of function being called. See "Types of Functions" on page 105 for more information.

Note: The error code in the environment block is not reset until another error occurs or a Boolean-type request is entered that returns TO2_IS_FALSE.

For more information about error codes, see the *TPF C/C++ Language Support User's Guide*.

Boolean Error Handling

By convention, TPF API return codes indicate successful or unsuccessful completion by returning positive or zero values, respectively. This convention is violated somewhat by TPFCS API functions returning positive or zero values for true or false results, respectively.

To distinguish an unsuccessful API return code from a Boolean zero result, use the `TO2_getErrorCode` function call for error value retrieval. A zero error code value indicates a successful return of TO2_IS_FALSE while a nonzero error code value indicates an error return from the Boolean API function.

The following logic might be used in an application program to test for an error on a Boolean function:

1. Call a Boolean TPFCS API; for example:

   `TO2_isEmpty`

   If a value of TO2_ERROR (0) is returned, this indicates that either an error has occurred or a value of TO2_IS_FALSE has been returned. If a value of TO2_IS_TRUE is returned, this indicates that no error has occurred.
2. Call the following TPFCS API to retrieve the error code value:

   `TO2_getErrorCode`

   If a value of TO2_IS_FALSE (0) is returned, this indicates that no error has occurred. All other values returned from `TO2_getErrorCode` indicate the error that occurred.
3. If an error has occurred, you can optionally call the following TPFCS API to retrieve the error code message text for the returned value:

   `TO2_getErrorText`

Example of a Returned Error Code

When an error occurs in the attempt to process a function, TPFCS sets an error code in the environment block and returns a `0` to the application program. The following example shows how an application can check to see if an error occurs when calling a Boolean TPFCS function.

```
#include <c$to2.h>                /* Needed for TO2 API Functions     */
#include <stdio.h>                /* APIs for standard I/O functions  */
TO2_PID             cursor;
TO2_ENV_PTR         env_ptr;
TO2_ERR_CODE        err_code;     /* TO2 error code value             */
TO2_ERR_TEXT_PTR    err_text_ptr; /* TO2 error code text pointer      */
   ⋮
/**********************************************************************/
/* Are there more elements after the current one?                     */
/**********************************************************************/
if (TO2_more(&cursor,env_ptr) == TO2_ERROR)
{
   err_code = TO2_getErrorCode(env_ptr);
   if (err_code != TO2_IS_FALSE)
   {
      printf("TO2_more failed!\n");
      err_text_ptr = TO2_getErrorText(env_ptr, err_code);
      printf("The error is: %s\n", err_text_ptr);
   }
   else
      printf("There are no more elements after the current element.\n");
}
else
      printf("There are more elements after the current element.\n");
```

Data Store Application Dictionary

To access a collection, the application must determine the PID of the collection. The recommended approach is to place the PIDs of data store anchor collections in the data store application dictionary and assign a symbolic name to each one as the key. This dictionary, which has the preassigned name of DS_USER_DICT, is accessed by establishing an environment for the target data store and using the `TO2_...DSdict...` type functions. For example, you can use the `TO2_atDSdictNewKeyPut` function to store the PID of an anchor in the dictionary. You can retrieve the PID of an anchor with the `TO2_atDSdictKey` function. The dictionary uses EBCDIC keys of 64 bytes with data elements of up to 1000 bytes.

Note: When PIDs are stored in collection elements, a recoup index must be established and associated with that collection, including the application dictionary.

The TPF dictionary (the application dictionary of TPFDB, the base TPFCS data store) can be accessed with a special set of functions (`TO2_...TPF...`). This allows all applications access to a common dictionary without needing to establish an environment for a particular data store. A possible use for this dictionary is to associate applications with data stores.

Note: The data store system dictionaries, accessed with the `TO2_...DSsystem...` and `TO2_...TPFsystem...` type functions, are used by the TPFCS system and are not intended to be used by applications.

Application Startup Examples

The examples in this section describe the steps that could be followed for initial population of a data store and for application startup.

Initial Population of a Data Store

When a data store is first created, the startup flow for an application that initially populates the data store could be as follows:

1. Create the environment using the target data store name on the `TO2_createEnv` function.
2. Create a recoup index for the application dictionary using the `TO2_createRecoupIndex` function.
3. Add a recoup index entry to the index describing how PIDs are stored in the dictionary using the `TO2_addRecoupIndexEntry` function.
4. Get the PID of the application dictionary using the `TO2_getDSdictPID` function.
5. Associate the recoup index with the dictionary by using the `TO2_associateRecoupIndexWithPID` function (see *TPF Database Reference* for more information).
6. Create an anchor collection using a `TO2_create...` function.
7. Store the PID of the anchor collection in the application dictionary by using the `TO2_atDSdictNewKeyPut` function. The key could be a symbolic name for the collection or any other value.
8. Optionally assign a name to the collection so that the collection can be easily accessed with the browser functional messages by using the `TO2_defineBrowseNameForPID` function. Consider using the same symbolic name used to store the PID in the application dictionary.

Optionally, you can associate particular applications with this data store by doing the following:

1. Create an environment using TPFDB as the data store name.
2. For each application that will use this data store, store the data store name in the TPF application dictionary using the `T02_atTPFNewKeyPut` function with the name of the application as the key.
3. Enter a `T02_deleteEnv` call to delete the environment.

Application Startup Flow

The startup flow for a typical application could be as follows:

1. Create the environment using the target data store name on the `T02_createEnv` function.
2. Access the TPF application dictionary to read the data store name of this application using the `T02_atTPFKey` function.

   Make the following assumptions:

   - The application uses the name of the application as the key when accessing the dictionary.
   - Another application puts the element into the dictionary (using the `T02_atTPFNewKeyPut` function) before this application was started.
3. Enter a `T02_deleteEnv` call to delete the TPFDB environment because it is not needed again to access TPFDB.
4. Create a new environment using the retrieved data store name.
5. Retrieve the PID of an anchor collection from the application dictionary by using the predetermined key on the `T02_atDSdictKey` function.
6. Continue with the remainder of the application processing.
7. When processing has been completed, enter a `T02_deleteEnv` call to delete the environment.

Types of Functions

The provided functions that make up the collection library are divided into three types:

- Collection functions

  A collection is a related group of elements organized within a data store. Collections are created by applications and may be temporary or persistent. The collection APIs allow collection creation and deletion and element manipulation and interrogation. Collections include abstractions such as array, set, bag, and key sorted set. Most TPFCS functions are *atomic*; that is, the element is read into storage, managed, and removed from storage for each function call.
- Cursor functions

  Cursors provide you with convenient methods for accessing and iterating through the elements stored in the collections. The cursor APIs allow element manipulation and interrogation. Cursors can also be used to lock collections to prevent other users from updating a collection while you are accessing the elements in the collection. Furthermore, cursors allow parts of collections to stay in memory so that some repetitive or consecutive accesses do not require data to be constantly read from or written to DASD. It also allows you to use alternate key paths.
- Auxiliary functions

  The auxiliary functions include dictionary, browser, and other miscellaneous functions:

- Dictionary functions are used to access the data store and TPF dictionaries. This allows applications to access a collection without having to know the actual PID of the collection.
- Browser functions allow you to do tasks such as the following:
  - Display information about a collection
  - Display the contents of a collection
  - Delete or reclaim a collection
  - Capture and restore a collection
  - Validate the integrity of a collection
  - Reconstruct a collection that has errors.

  For more information about TPFCS functions, see the *TPF C/C++ Language Support User's Guide*.

Returned Data Structures

When TPFCS returns an element in response to a `TO2_at`, `TO2_atCursor`, `TO2_peek`, or other similar function (except for `TO2_atRBA`), it returns a data structure that resides in a private heap storage area. The calling routine must use the `free` function call to free the buffer once it has completed processing the returned data. When the function call includes the pointer to a buffer (for example, a `TO2_atWithBuffer`, `TO2_atCursorWithBuffer` or a `TO2_peekWithBuffer` function call), the data is also returned in the buffer.

The normal return from these types of APIs is a pointer (TO2_BUF_PTR) to a structure (buffer) of type TO2_BUF_HDR. The structure of this buffer has five fields:

Field	**Description**
spare	Type long, reserved for IBM use.
updateSeqNbr	Type long, update sequence counter value.
dataL	Type long, length of the data.
spare	Type long, reserved for IBM use.
data	Array of `char`, the beginning of the actual data. The data does not contain the key.

Collection APIs

TPFCS provides APIs for performing certain operations on a collection. This section describes the types of operations that can be performed on a collection.

Creating and Deleting Collections

When you create the collection instance, you do so by calling the appropriate `TO2_create...` function for the instance you want. There is a unique create function for each TPFCS collection type. The `TO2_create...` function returns the PID assigned to the collection.

Several different options can be selected when creating collections using the `TO2_create...WithOptionList` type APIs and specifying customized data definitions. These options include:

- Collection lifetime (persistent long-term, persistent short-term, or temporary)
- Whether or not the collection is shadowed
- What recoup index (if any) is associated with the collection.

For more information about collection lifetimes, see *TPF Database Reference*.

You can delete collections dynamically by using the `TO2_deleteCollection` function. After you enter the `TO2_deleteCollection` function, the collection is either marked for deletion or actually deleted from the database and cannot be accessed by other TPFCS functions. For persistent short-term collections and temporary collections, the deletion always takes place immediately. For persistent long-term collections, the deletion is controlled by the data store characteristics set with the ZOODB command. A persistent long-term collection marked for deletion can be reclaimed by entering the `TO2_reclaimPID` function in the time period between the `TO2_deleteCollection` request and the time the actual deletion occurs (48 hours).

Accessing and Modifying Collections

You can perform the following operations to modify a collection:

Modification	**Operation**
Adding elements	Use the `TO2_add` function and its variants.
Finding and retrieving elements	Use the `TO2_at` function and its variants.
Updating elements	Use the `TO2_atPut` function and its variants.
Removing elements	Use the `TO2_remove` function and its variants.

Adding Elements

For nonkeyed collections, the `TO2_add` function places the element identified by its argument into the collection. For keyed collections, the `TO2_atNewKeyPut` function places the element identified by its key into the collection. See *TPF C/C++ Language Support User's Guide* for more information about the `TO2_add` and `TO2_atNewKeyPut` functions. For binary large objects (BLOBs), you can operate on elements using the `TO2_atRBAPut` function. In general, you can copy one collection to another collection that is initially empty by iterating through the elements of the first collection and calling `TO2_add` or `TO2_atNewKeyPut` with each element as an argument.

For sequence collections, elements can be added at a given position using `TO2_addAtIndex`.

Table 13 describes how the `TO2_add` and `TO2_atNewKeyPut` APIs function with unique and nonunique collections:

Table 13. Adding Elements to a Collection

Type of Collection	Description
Unique collections without keys	The `TO2_add` function will not add an element that is equal to an element that is already in the collection.
Nonunique collections without keys	The `TO2_add` function adds elements regardless of whether they are equal to any existing elements.
Unique collections with keys	The `TO2_atNewKeyPut` function will not add a key that is equal to a key that is already in the collection.
Nonunique collections with keys	The `TO2_atNewKeyPut` function adds keys regardless of whether they are equal to any existing keys.

Finding and Retrieving Elements

You can find an element without using a cursor by searching for it based on position or key by using the `TO2_at`, `TO2_atKey`, and `TO2_atRBA` function. In arrays, BLOBs, keyed logs, logs, or sequence collections, elements are addressed by their index. The index of the first element is always 1; that is, they are 1-based. Elements in these types of collections are accessed using the `TO2_at` API (data in BLOBs are accessed using the `TO2_atRBA` API). For keyed collections, an element can be addressed by its key using `TO2_atKey`. For collections without unique keys, such as key bags, only the first element with the given key can be explicitly addressed. In collections such as bags, sets, and sorted bags, elements have no address component and, therefore, cannot be individually selected except through the use of cursors (see “Cursors” on page 109). Once you have found and retrieved the element, you can update it and then replace it.

Updating Elements

It is possible to modify collections by updating the value of an element occurrence. The maximum length of the updated element cannot be changed; however, elements can be shorter than the specified maximum length.

The key or element value that must be preserved is called the *positioning property* of the element in the given collection. For nonkeyed collections with element equality (bags and sets), an update function is not provided. *Element equality* is a condition where two elements in a single collection are equal in length and equal in bit sequence. To change an element for a nonkeyed collection, the old element must be removed and the new element added to the collection.

For sorted collections that are organized according to an element property (the sort field), if the update function changes this element property, the old element is automatically deleted and the new updated element is automatically placed in its correct position in the collection.

Arrays, BLOBs, keyed logs, logs, and sequence collections do not have a positioning property; that is, instead of having a key that can be preserved or element value, they have a logical position that is independent of the content of the element. Element values in these collections can be changed freely using the `TO2_atPut` function.

Update sequence counter: As a form of database integrity for those collections that allow element updating, an update sequence counter is stored in each element in a given collection and optimistic concurrency is enforced on the element during update processing. For more information, see “Optimistic Concurrency (Update Sequence Counter)” on page 115.

Removing Elements

To remove elements from a collection, you can either use a specific remove-type function (`TO2_removeIndex`, `TO2_removeValue`, or `TO2_removeKey`) or you can remove an element that is pointed to by a given cursor (see “Using Cursors for Locating, Accessing, and Removing Elements” on page 110).

There is an important difference between element values and element occurrences. An element value may, for nonunique collections, occur more than once. The basic remove-type function removes ***only*** the first occurrence of an element. If you want to remove all occurrences of the elements in a collection that have a given property, use the `TO2_removeValueAll` function.

For collections with key equality or element equality, removal functions remove one or all occurrences of a given key or element. Sequence collections provide functions for removing an element at a given index.

Cursors

A *cursor* is a nonpersistent internal structure associated with a collection that is used to reference an element in the collection. Cursors are used for the following:

- To iterate through collections

  The cursor provides methods that allow an application program to move through a collection one element at a time without needing keys or indexes.
- To establish locks on collections

  When a locking cursor is created for a particular collection, a lock is placed on the entire collection so that only the ECB that created the cursor can update the elements in the collection.
- To improve processing efficiency

  It is more efficient to access multiple elements in a collection using a cursor. Most TPFCS functions are *atomic*; that is, the element is read into storage, managed, and removed from storage for each function call. This type of processing requires much overhead. With cursors, once a collection is read into storage, it remains there for the life of the cursor. The cursor is temporary and must be deleted by the application program or when the ECB exits.

For more information about cursor APIs, see the *TPF C/C++ Language Support User's Guide*.

Cursors can be used with any collection. If you use the cursor APIs to add or remove elements, cursor positioning remains valid unless an error occurs. If you use the collection (noncursor) APIs on collections that have a cursor associated with them to add or remove elements, cursor positioning might not be valid. One of the following conditions occurs:

- The cursor points to the same element.
- The cursor points to a different element.
- The cursor no longer points to an element of the collection.
- The cursor is marked as not valid.

Positioning might not be valid because a cursor points to a particular element position within a collection, not to an element itself. If collection APIs are used to add or remove elements in a collection that has a cursor pointing to a specific element, the elements in the collection may shift position and the cursor may no longer be pointing to same element. For example, if the cursor last pointed at the fifth position and an element was inserted after the first position, the cursor would be pointing at the same position when positioning is checked again. However, this position would now be referencing what was originally the fourth element.

When the cursor is used again, the positioning is checked and, if the cursor no longer points to an element or to an element that matches the current positioning information of the cursor, the cursor is marked as not valid and a request to reposition the cursor to a specific element is required before it can be used again. Note that elements in collections that allow multiples do not have a uniqueness among themselves. As a result, it is possible for the current positioning information of the cursor to match an element that is not the original element to which the cursor was pointing.

Initializing a Cursor

When a cursor is created, the cursor does not have a default position. A positioning-type operation (such as `TO2_first` or `TO2_locate`) is required before the cursor can be used to access the target collection. Whenever an error code indicating that the cursor is not valid (TO2_ERROR_CURSOR) is returned, it signifies that the cursor function has lost position either because the collection has been changed by the current user, some other user (only for dirty-reads), or because of a TPFCS problem. When you see this error code, try to reposition the cursor with a positioning-type operation on the cursor.

TPFCS provides the following functions to construct a cursor for a given collection:
- `TO2_createCursor`
- `TO2_createReadWriteCursor`.

If the position of the element changes, TPFCS attempts to reposition the cursor to point to the subject element. If TPFCS is unable to do so, the cursor is not valid. This occurs because the cursor refers only to the position of the element and not to the element itself.

If you add or remove elements from a collection while you are iterating over a collection (except by using the cursor doing the iteration), all elements may not be visited once.

Using Cursors for Locating, Accessing, and Removing Elements

Cursors provide a basic mechanism for accessing elements of collections. For each collection, you can define one or more cursors and you can use these cursors to access elements. Collection functions such as `TO2_locate`, `TO2_setPositionIndex`, and `TO2_setPositionValue` use cursors to locate and access specific elements. You can then access the actual element by using cursor functions such as `TO2_atCursor`, `TO2_atCursorPut`, or `TO2_remove` (see “Removing Elements” on page 108 for more information).

Note: Cursor functions specify the PID of the *cursor*, not the PID of the *collection*.

Using Cursors with Alternate Key Paths

A key path is used to determine the order in which some collections are traversed. There are two types of key paths: primary and alternate. When a collection is first created, the primary key path of the collection is used by default for searching and accessing data. You can use alternate key paths only with cursors.

You can use the `TO2_setKeyPath` function to override the default setting or any previous `TO2_setKeyPath` calls to specify an alternate key path. A maximum of 16 alternate key paths (in addition to the primary) can be defined for each collection. When the `TO2_setKeyPath` function is issued, the position of the cursor must be reestablished by using one of the positioning functions such as `TO2_first`. See “Key Path Support” on page 116 for more information.

Cursor Positioning

The first and last elements in a collection are based on what type of collection it is. The following sections describe what the first and last elements are according to the type of collection.

First Element in a Collection

The first element in a collection is based on the type of collection:

- For log and keyed log collections, it is the oldest element.
- For other ordered but nonkeyed collections (array, BLOB, and sequence), it is the element at index 1.

- For other collections with a key or sort field (key sorted set, key bag, key set, key sorted bag, sorted bag, and sorted set), it is the element with the lowest key or sort field value.
- For all other collections (bag and set) without an explicit ordering, it is determined by the implementation (that is, randomly).
- When the cursor is positioned at the first element in the collection and a `TO2_cursorMinus` request is issued, a TO2_ERROR_EODAD error is returned and the cursor will be marked as not valid.

  Note: With the exception of sequence collections, elements cannot be added before the first element by issuing a `TO2_cursorMinus` function to position the cursor before the first element.

  Table 14 summarizes what happens when a cursor is positioned at the start of a collection (`TO2_first` returns TO2_IS_TRUE).

Table 14. Cursor Positioned at the Start of a Collection

Function Name	Result	Cursor Status
`TO2_addAtCursor`	Adds the element as the first element.	Positioned to point at the inserted element.
`TO2_atCursor`	Reads the first element.	Unchanged.
`TO2_atCursorPut`	Updates the first element.	Unchanged.
`TO2_cursorMinus`	TO2_ERROR_EODAD	Not valid.
`TO2_cursorPlus`	Updates the cursor position (or returns EODAD if the collection contains only one element).	Positioned to point at the next element.
`TO2_remove`	Removes the first element.	Positioned to point at the next element, which is now the first.

Last Element in a Collection

The last element in a collection is based on the type of collection:

- For log and keyed log collections, it is the newest element.
- For other ordered but nonkeyed collections (array, BLOB, and sequence), it is the element with the highest index value.
- For other collections with a key or sort field (key bag, key set, key sorted bag, key sorted set, sorted bag, and sorted set), it is the element with the highest key or sort field value.
- For all other collections (bag and set) without an explicit ordering, it is determined by the implementation (that is, randomly).
- The following applies to all collection types ***except*** for arrays and BLOBs:
  - When the cursor is positioned at the last element in a collection and a `TO2_cursorPlus` function call is issued, the cursor will be positioned at the end of the collection. If another `TO2_cursorPlus` function call is issued, it will receive a TO2_ERROR_EODAD return code and the cursor will not be valid. Any further cursor operations will receive a TO2_ERROR_CURSOR until an explicit positioning cursor operation is issued, such as `TO2_first` or a `TO2_last`.

    Table 15 on page 112 summarizes what happens when a cursor is positioned at the end of a collection (`TO2_atEnd` returns TO2_IS_TRUE).

Table 15. Cursor Positioned at the End of a Collection

Function Name	**Result**	**Cursor Status**
`TO2_addAtCursor`	Adds the element as the last element.	Positioned to point at the inserted element.
`TO2_atCursor`	TO2_ERROR_EODAD	Unchanged.
`TO2_atCursorPut`	TO2_ERROR_EODAD	Unchanged.
`TO2_cursorMinus`	Successful.	Positioned at the last element.
`TO2_cursorPlus`	TO2_ERROR_EODAD	Not valid.
`TO2_remove`	TO2_ERROR_EODAD	Unchanged.

- Arrays and BLOBs are handled differently because they can grow by logically adding NULL entries. A `TO2_cursorPlus` request can continue to advance beyond the end of the collection without receiving a TO2_ERROR_EODAD error. If a `TO2_atCursor` or a `TO2_remove` request is issued when the cursor has advanced beyond the end of the collection, a TO2_ERROR_EODAD will be returned. If a `TO2_atCursorPut` request is issued, the element will be added to the collection at the current position of the cursor and NULL entries will be logically added from the original end of the collection to the element before the added element.

Determining the End of the Collection

The end of the collection is defined as the position after the last existing element in the collection. For collections without an explicit ordering, this position is determined by the implementation (that is, randomly). In the following example, x points to the first element, y points to the last element, and z points to the end of the collection.

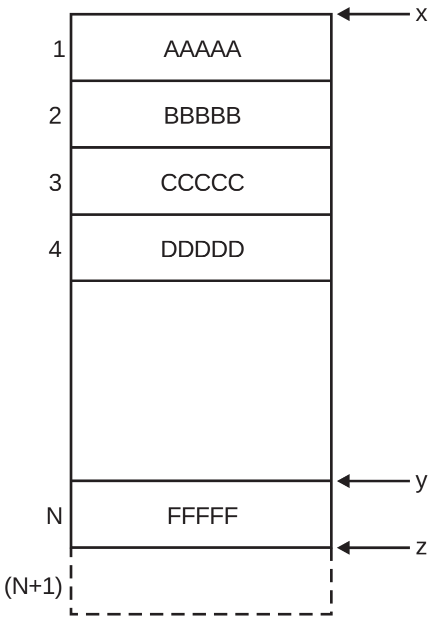


Figure 16. Collection with N Elements

Table 16 on page 113 relates to Figure 16 and provides a summary of cursor returns for the `TO2_first`, `TO2_atLast`, and `TO2_atEnd` functions.

Table 16. Cursor Returns

Cursor Position	**TO2_First**	**TO2_atLast**	**TO2_atEnd**
x	TO2_IS_TRUE	TO2_IS_FALSE	TO2_IS_FALSE
y	TO2_IS_FALSE	TO2_IS_TRUE	TO2_IS_FALSE
z	TO2_IS_FALSE	TO2_IS_FALSE	TO2_IS_TRUE

Rules for Cursor Movement and Positioning

TPFCS observes the following rules for cursor movement and positioning:

- If an error occurs while the cursor attempts to perform a positioning request, the cursor will be marked as not valid and the error will be returned to the caller (except as noted for `TO2_locate`).
- The `TO2_remove` function call will cause the element where the cursor is positioned to be removed (zeroed if it is an array or BLOB collection). The cursor will be positioned to point at the following element unless it was the last element in the collection, in which case it will be positioned at the end of the collection.
- `TO2_locate`
  - For key sorted bag, key sorted set, sorted bag, and sorted set collections, a `TO2_locate` request that returns a TO2_ERROR_LOCATOR_NOT_FOUND error will position the cursor to point at the next higher key or sort field.
  - For key sets and key bags, a `TO2_locate` request that returns a TO2_ERROR_LOCATOR_NOT_FOUND error will cause the cursor to be not valid.
- For bags and sets, `TO2_locate` implicitly calls `TO2_setPositionValue` and returns a TO2_ERROR_LOCATOR_NOT_FOUND error that will cause the cursor to be not valid.
- For key bag, key set, key sorted bag, key sorted set, sorted bag, and sorted set collections, `TO2_setPositionValue` implicitly calls `TO2_locate`.
- `TO2_locate` and `TO2_setPositionValue` are not supported for keyed log collections.

Iterating over Collections

Iterating over all or some elements of a collection is a common operation. The collection library gives you two methods of iteration:

- Using cursors
- Using the `TO2_allElementsDo` function with your own function.

Ordered collections, such as array and sorted set, have a well-defined ordering of their elements. Unordered collections, such as bag and set, have no defined order in which the elements are visited in an iteration; however, for these types of collections, each element is visited exactly once. Similarly, unique collections, such as set and key sorted set, do not allow multiple elements with the same key or value. Nonunique collections, such as bag and key sorted bag, allow duplicate values and keys; again, for these types of collections, each element is visited exactly once.

Iteration Using Cursors

Do not add or remove elements from a collection while you are iterating over a collection except by using the cursor that is doing the iterating or all elements may not be visited once. See “Removing Elements” on page 108 for more information about removing elements.

Cursor iteration can be done with a *for loop*. Consider the following example:

```
#include <c$to2.h>                /* Needed for TO2 API functions   */

TO2_PID       myCollection;
TO2_PID       myCursor;
TO2_ENV_PTR   env_ptr;
MyIntElement *currentElementPtr;

TO2_BUF_PTR   bufferPtr;
   ⋮
/**********************************************************************/
/* Access each of the elements in the collection...                 */
/**********************************************************************/

TO2_createCursor(&myCollection, env_ptr, &myCursor);

for (TO2_first(&myCursor, env_ptr),
     TO2_more(&myCursor, env_ptr),
     TO2_cursorPlus(&myCursor, env_ptr))
{
   ⋮
   if ((bufferPtr = TO2_atCursor(&myCursor,env_ptr)) == TO2_ERROR)
   {
     printf("TO2_atCursor failed!\n");
     process_error(env_ptr);
   }
   else
   {
       /* work with currentElement using currentElementPtr.*/
       currentElementPtr = (MyIntElement *)bufferPtr->data;
   ⋮
   }


   free(bufferPtr);      release returned buffer
}
   ⋮
TO2_deleteCursor(myCursor, env_ptr);
```

In this example, a cursor is created for myCollection. The loop is initialized by pointing the cursor to the beginning of the collection. The loop then iterates over all elements stored in the collection. The data component of each element is accessed and manipulated. The loop ends when there are no more elements remaining to process in the collection. Finally, the cursor is deleted.

Note: This code example does not show any environment tests for possible errors.

TO2_allElementsDo

Cursor iteration has two possible drawbacks:

- For unordered collections, the explicit notion of an (arbitrary) ordering may not be desirable for reasons of style. For example, it could mislead you (or other programmers) into perceiving or exploiting an order where, in fact, the order does not exist or is not guaranteed.
- Iteration on a collection might be done more effectively (because of a shorter path length) by the collection itself than by the application using cursors and explicit positioning calls.

The collection library provides the `TO2_allElementsDo` function, which addresses both drawbacks by calling a user-specified function that is applied to all elements. The user-specified function returns a value that is used internally to indicate the continuation or ending of the iteration. For collections with order, the function is applied in this order. Otherwise, the order is not specified.

Additional arguments that are needed for the iteration can be passed as an extra parameter list on the `T02_allElementsDo` function.

Using Cursors for Locking Collections

Cursors are also used to prevent concurrent updating of a collection while you are accessing elements in the collection.

The following summarizes TPFCS operations involving the two different types of cursors:

- When a nonlocking cursor is used on a collection, the same ECB can:
  - Perform noncursor reads and writes
  - Perform cursor reads
  - Create other nonlocking cursors
  - Create a locking cursor.

  A different ECB can also do all of the above.
- When a locking cursor is used on a collection, the same ECB can:
  - Perform noncursor reads and writes
  - Perform cursor reads and writes
  - Create other nonlocking cursors.

The same ECB cannot create another locking cursor. A different ECB can do all of the above and it can also create another locking cursor, although the request will be deferred until the original locking cursor is deleted.

Concurrency Controls

TPFCS provides three levels of concurrency control:

- None (using nonlocking cursors)
- Optimistic (using update sequence counters)
- Pessimistic (using locking cursors).

None (Nonlocking Cursor)

The first type of concurrency uses a nonlocking cursor to read elements in a collection without creating any type of interlock on the target collection. For this reason, it is considered a *dirty-read* cursor. A subsequent request to access the collection by a locking cursor will be successful and an exclusive lock will be placed on the collection by the new cursor. However, the nonlocking cursor will still be able to read elements in the collection.

Optimistic Concurrency (Update Sequence Counter)

To provide additional database integrity for those collections that allow element updating, an update sequence counter is stored in each element in a given collection and optimistic concurrency is enforced on the element during update processing. Optimistic concurrency allows you to read a collection and update it without exclusive access to the collection. When an element is to be updated, it must first be read from the collection with the returned sequence counter saved. The functions used to update elements in a collection require that the application provide the expected value for this counter. TPFCS increments the counter whenever a collection element is updated. If the collection is updated by some other user, your current update request will fail because the value passed by the application as input to the function does not match the update sequence counter embedded in the element. The collection must be retrieved again for a successful update to take place.

Pessimistic Concurrency (Locking Cursor)

Pessimistic concurrency uses a locking cursor to create an exclusive lock on the collection. No other locking cursor in the same ECB is able to create a lock on the collection and the attempt is rejected with an error return code. If another ECB attempts to access the collection using a locking cursor, the ECB is forced to wait until the first ECB either deletes its cursor or exits. This is called a *dirty-read cursor* because there is no guarantee that two back-to-back reads can read the same element. However, the nonlocking cursors are still able to read the collection.

Note: Locking cursors do not prevent a dirty read from being done by some other user.

Dirty-Reader Protection

TPFCS provides dirty-reader protection by using the FILNC macro to file records in a sequence to make sure that an updated or new record is on the DASD surface before filing the record that points to the updated record. This is done to ensure that another user who is attempting to read the collection using a nonlocking cursor will be able to follow a whole chain. If the updates were filed in the wrong order, it might be possible for the reader to follow a chain with holes in it or the chain could point to records that were not even part of the collection; for example:

> Record A contains a pointer to record B and now TPFCS has to insert record C between record A and B. If TPFCS filed record A with the new pointer to record C before filing record C, it would then be possible for a reader to read record A and then attempt to read record C before it had been filed. To prevent this, TPFCS dirty-reader protection will make sure that record C is filed first by using the FILNC macro, and then record A will be filed.

For more information about the FILNC macro, see *TPF General Macros*.

Key Path Support

Key path support enables you to search for and access data in a collection by using the value of a specified data field or key.

TPFCS supports two types of key paths, primary and alternate, for the following persistent keyed and sorted collections:
- Key bags
- Key sets
- Key sorted bags
- Key sorted sets
- Sorted bags
- Sorted sets.

Note: Sorted bag and sorted set collections do not have primary keys; they only have sort fields.

Primary key paths are sorted in ascending binary value by the primary key, and they can be either unique or nonunique depending on the collection type. *Alternate* key paths support nonunique key path fields (keys) and are sorted in ascending binary value based on the key path field. A maximum of 16 alternate key paths (in addition to the primary key path) can be defined one at a time for each collection.

Atomic functions (APIs) such as `TO2_atKey`, `TO2_atKeyPut`, `TO2_atNewKeyPut`, and `TO2_removeKey` use only the primary key path for accessing data. You can use alternate key paths only with cursors. The application program can use a cursor to access primary or alternate key paths. To use a key path, the application program creates a cursor by using either the `TO2_createCursor` or `TO2_createReadWriteCursor` function. Initially, the cursor uses the primary key path by default. The application can change the key path and assign an alternate key path to the cursor by using the `TO2_setKeyPath` function and specifying a key path name.

The application program can also issue a `TO2_setKeyPath` call with the TO2_PRIME_KEYPATH name to reset the cursor to use the primary key path.

While the key path build process is in progress, the TO2_ERROR_KEYPATH_BUILD_ACTIVE error code is returned. The specified key path is not usable until the build process has ended.

Adding Key Paths

Primary key paths are automatically established when a collection is created in a TPFCS database. Alternate key paths are added to a collection by using the `TO2_addKeyPath` function. Using the `TO2_addKeyPath` function, the application program specifies the name of the new key path and the displacement and length of the field in the data element that the key path will reference. If a particular data element is too short to contain the entire field, it will still be included in the key path as if bytes of zero were concatenated at the end for the remaining length of the key.

Key paths are built automatically by an asynchronous task if the collection already exists and contains data elements. The key path is not usable until the build process has completed successfully.

After the key path has been defined, TPFCS will automatically update it whenever the collection is updated. Therefore, collections with alternate key paths will take longer to update because they have to maintain the internal structures of every key path.

Removing Key Paths

To remove an alternate key path from the collection, the application program issues a `TO2_removeKeyPath` request specifying the name of the key path to remove. The `TO2_removeKeyPath` function deletes the key path from the collection and releases the resources of the key path back to the system.

Note: You ***cannot*** remove the primary key path from a collection.

Understanding Logical Record Caching

A *Logical record cache* is a set of system functions that allows an application to define and use memory caches. A *cache* is a hashed structure for holding information in *lookaside storage buffers*. A *lookaside storage buffer* is a temporary storage area where a copy of the data is saved to avoid refetching the data on every access. The data is retrieved from the temporary storage area instead of its permanent residence. The cache gives the application fast access to frequently used information without always needing to retrieve the information from an external storage device. The logical record caches are named caches that are subsystem shared, but the entries are subsystem unique. If two applications try to use the same name for their cache, the first call will create the cache and the second application will be connected to the created cache as long as the structure attributes of the create call for the second application are identical to the first application. Otherwise, the create call for the second application is returned with an error code.

Logical record caches are subsystem shared and either processor unique or processor shared. *Processor unique caches* contain information that pertains only to the processor where the information resides and does not need to be kept synchronous with the image of any other processor. *Processor shared caches* contain information that is shared across the processors in the complex so the information must be kept synchronous with the information contained in the caches for the other processors. In a processor shared cache, updating an entry will cause all other processors using the same named cache to have their representation of the information invalidated so that a CACHE_NOT_ FOUND return code is returned the next time the information is accessed.

Logical record cache support uses coupling facility (CF) cache support to maintain information synchronization between the same named caches residing in different processors in the complex. This synchronization is done through the cache name. At least one CF must be added to the locking configuration by using the ZCFLK ADD command before a logical record cache can use a CF. When a processor shared cache is created and a CF is available for use, a CF cache structure is created in the CF with the same name as the cache that is created. If the CF cache structure already exists, the processor is connected to the structure that already exists. The CF cache structure is a directory-only structure that is used to keep track of the entries that are in the cache of each processor. Through this mechanism, the processor copy of the cache is notified of an update to an entry by another processor, which causes the CF to invalidate the corresponding entry of the processor. CF cache structures are created with a disposition of DELETE (STRDISP=DELETE specified on the CFCONC macro). See *TPF Database Reference* for more information about CF cache support, CF cache structures, and structure persistence for CF cache structures. See *TPF Operations* for more information about the ZCFLK ADD command. See *TPF System Macros* for more information about the CFCONC macro.

The information held in the logical record cache must also be resident on an external storage device. A cache is lost over a processor IPL and must be reestablished by the application after an IPL occurs. There is no mechanism for preserving the contents of a cache over an IPL.

The logical record cache uses the database ID (DBI) in the entry control block (ECB) and the primary and secondary keys (which are passed on the function call) to identify entries in the cache. The DBI and the keys are hashed and the result is used to locate a specific entry in the cache. The primary and secondary keys can be from 1 to 256 characters in length. A cache must have a primary key length defined, but does not need a secondary key. The keys and the DBI are used to uniquely identify an entry in the cache. The length of the keys specified on the function calls must be less than or equal to the lengths specified on the create function call. Because the cache is hashed, an entry can only be found as long as the specified keys and DBI exactly match the keys on the entry. There is no way to perform a partial key search on the entries in the cache. When the cache is processor shared, the values of the keys and the DBI are hashed into a 14-byte hash name that is used to identify an entry to the CF. Because the CF and logical record cache support do not support synonyms for the 14-byte hash name, logical record cache support will not store two different entries that hash to the same 14-byte hash name. This is true whether the cache is using the CF or not.

When adding or updating an entry in the cache, it is the responsibility of the application to serialize the call. If the call is not serialized, it is possible for two ECBs on different central processing units (CPUs) in the same processor to try to add or update the same entry at the same time. If this occurs, logical record cache support adds the entry for one and then overlays it for the other. It is difficult to predict which CPU call will cause the information to be overlaid. This is also true for processor shared caches. To guarantee that the cache does not contain old information, the application should perform all adds and updates while maintaining an external serialization lock. Otherwise, it is possible for a cache invalidate to flow from one processor to another through the CF, and for the second processor to retrieve old information from the external storage device and add the information again before the first processor has completed updating the information.

The following are examples of caches the TPF system currently uses:
- File system directory cache
- File system i-node cache
- Domain Name System (DNS) Internet Protocol (IP) and host names.

Creating a Logical Record Cache

To create a logical record cache, the application starts the `newCache` function, passing the following as inputs:
- The name of the cache being created
- A pointer to a field where the token for the created cache will be stored
- The attributes of the information that will be stored in the cache.

See *TPF C/C++ Language Support User's Guide* for more information about the `newCache` function.

The following shows the function to create a logical record cache:

```
long    newCache(const void  *  cache_name,
                 cacheTokenPtr  cachetokenReturn,
                 const long     primaryKeyLength,
                 const long     secondaryKeyLength,
                 const long     dataLength,
                 const long     numberEntries,
                 const long     castoutTime,
                 const long  *  type_of_cache,
                 const long  *  reserved);
```

System Heap and the Hash Table

When logical record cache support creates a new cache, it resides in the system heap. The amount of system heap allocated is determined by the values passed on the `newCache` function for the primaryKeyLength, secondaryKeyLength, dataLength, and numberEntries parameters. The numberEntries parameter is used to determine the size of the hash table and the number of entries the cache must hold. Determine the minimum number of entries needed. If more entries are needed as shown by an unacceptable castout rate in the data collection reports, you can increase the value of the entries by entering the ZCACH command without changing the code. See *TPF Operations* for more information about the ZCACH command. See *TPF System Performance and Measurement Reference* for more information about the data collection reports.

To determine the size of the hash table, logical record cache support doubles the value passed for the number of entries and calculates the best prime number to use as the hash divisor. The resulting value becomes the size of the hash table. The size of an entry is calculated by adding the values for the primaryKeyLength, secondaryKeyLength, and dataLength parameters plus some TPF system overhead for chaining. Therefore, the amount of system heap allocated is determined by adding the following values together:

- The size of the hash table
- The size of the entry area (`entry size * value of the numberEntries parameter`)
- A cache header area.

See *TPF System Generation* for more information about the system heap.

Cache Name

The cache name is used to define the cache and to allow other applications to connect to the cache by issuing their own `newCache` function call and specifying the same cache name. The first application to issue the `newCache` function for a specific cache name will cause the cache to be created and its attributes set. The `newCache` function call with the same cache name must pass parameters that are checked against the attributes of the cache and, if they are the same, the caller is connected to the cache. If the parameters and the attributes do not match, an error code is returned to the caller.

The cache name:

- Must be 4 to 12 alphanumeric characters
- Must begin with an alphabetic character
- Can contain the following special characters: at sign (@), dollar sign ($), or underscore (_)
- Must be padded to the right with blanks if the cache name is less than 12 characters.

Note: All cache names beginning with the letter **I** are reserved for IBM use. Additionally, all cache names beginning with the letters **tpf** in uppercase, lowercase, or mixed case are reserved for IBM use. See *TPF Programming Standards* for more information about naming conventions.

The following are examples of valid cache names:

- CacheBSS1
- BSS_cache

- US_Cities
- Denver@12

The following are examples of cache names that are not valid:

Aca	Is less than 4 characters in length.
AnameThatIsTooLong	Is greater than 12 characters in length.
TPFcache2	Begins with IBM reserved letters.
Icache6	Begins with an IBM reserved letter.
cache15*	Contains an asterisk (*), which is an unsupported character.

cacheToken Value

The cacheToken is a value returned from the `newCache` function. The cacheToken value is returned in the field pointed to by the cacheTokenPtr parameter and is used for all other function calls to logical record cache support to identify the specific cache to act on. Whenever a specific cache is to be read, updated, or deleted, the cacheToken value identifies the cache. The passed token is validated by logical record cache support and, if it is not valid, an error code is returned to the caller and the function is not performed. The returned token can be saved and passed on all other iterations of code using the cache, or a `newCache` function can be started on every iteration of the code to retrieve the cacheToken value.

Examples

The following example shows how to define and use a cacheToken value:

```
#include <c$cache.h>
cacheToken      myCache;
```

Then set the cacheToken by issuing the following.

```
newCache (cache name,     /* name of the cache                     */
          &myCache,       /* address of where to store the token  */
          ...);           /* the remainder of the parameters      */
```

The following shows how cacheToken is used on a `readCacheEntry` function.

```
readCacheEntry(&myCache,    /* address of the token to use     */
             ......);       /* the remainder of the parameters */
```

Processor Unique and Processor Shared Caches

Logical record caching supports processor unique and processor shared caches. The cache type is passed as a parameter on the `newCache` function.

Examples

The following example shows how to pass the cache type on the `newCache` function.

```
/* To create a processor unique cache   */

   #include <c$cache.h>
   char  cacheType = Cache_ProcQ;  /* processor unique cache  */

   newCache(cache name,      /* name of the cache                    */
            &myCache,        /* address of where to store the token */
            ...              /* other parameters                    */
            &cacheType,      /* address of the cache type value     */
            NULL);           /* reserved set to NULL                */

/* To create a processor shared cache  */
```

```
   #include <c$cache.h>
   char  cacheType = Cache_ProcS; /* processor shared cache    */

   newCache (cache name,      /* name of the cache                   */
             &myCache,        /* address of where to store the token */
             ...              /* other parameters                    */
             &cacheType,      /* address of the cache type value     */
             NULL);           /* reserved set to NULL                */
```

castOutTime Value

The castOutTime parameter specifies the default time, in seconds, that an entry can exist in cache before it is considered old and must be replaced. This value is only used for processor unique caches or for processor shared caches that are not being managed using a CF to handle entry invalidations from other processors. If the cache is processor shared and connected to a CF cache structure, the castOutTime value is ignored. If there are no CFs available for use by the cache, the cache is still defined to be processor shared but operating in *local* mode and the castOutTime value is used.

The following example shows how to create a processor shared cache with primary and secondary keys.

```
#include <c$cache.h>
  char         cacheNameS[12] = "Shared_Cache";   /* cache name */
  char         cacheNameU[12] = "Unique_Cache";   /* cache name /
  cacheToken   myCacheShared;                /* where to save cacheToken   */
  cacheToken   myCacheUnique;                /* where to save cacheToken   */
  long         primaryKeyLgh = 255;          /* 255 byte primary key       */
  long         secondaryKeyLgh = 32;         /* 32 byte secondary key      */
  long         dataLgh = 48;                 /* 48 bytes of data           */
  long         numbEntries = 200;            /* 200 entries                */
  long         castOutTime = 60;             /* cast entry after 60 seconds*/
  char         cacheType = Cache_ProcS;      /* processor shared cache     */

  if( newCache ( cacheNameS,                 /* name of the cache                   */
                 myCacheShared,              /* address of where to store the token */
                 primaryKeyLgh,              /* maximum primary key length          */
                 secondaryKeyLgh,            /* maximum secondary key length        */
                 dataLgh,                    /* maximum data length                 */
                 numbEntries,                /* number of entries in cache          */
                 castOutTime,                /* cast out Time value                 */
                &cacheType,                  /* address of cache type value         */
                 NULL )                      /* reserved set to NULL                */
                           !=CACHE_SUCCESS)  /* successful create                   */
 {
      printf("error creating Shared_Cache"); /* write error msg  */
      exit(1);                               /* and exit         */
 }
```

The following example shows how you can create a processor unique cache with only primary keys and no castout time.

```
cacheType = Cache_ProcQ;         /* processor unique cache  */

if( newcache ( cacheNameU,                /* name of the cache                   */
               &myCacheUnique,            /* address of where to store the token */
               primaryKeyLgh,             /* maximum primary key length          */
               NULL,                      /* no secondary keys                   */
               dataLgh,                   /* maximum data length                 */
               numbEntries,               /* number of entries in cache          */
               NULL,                      /* cast out Time value                 */
               &cacheType                 /* address of cache type value         */
               NULL )                     /* reserved set to NULL                */
                       !=CACHE_SUCCESS)   /* successful create                   */
```

```
{
    printf("error creating Unique_Cache");   /* write error msg     */
    exit(1);                                 /* and exit            */
}
```

Reading an Entry from a Logical Record Cache

To read an entry from a logical record cache, the application starts the `readCacheEntry` function, passing the following as inputs:

- A pointer to the returned cacheToken value
- The primary and secondary key values and lengths
- A pointer to a buffer to hold the data that is found and the length of the buffer.

See *TPF C/C++ Language Support User's Guide* for more information about the `readCacheEntry` function.

The primary and secondary keys must exactly match the primary and secondary keys that are used to add the entry to the cache. This includes both the content of the keys and their lengths. Additionally, the ECB must have the same DBI as the ECB that is used to add the entry. If the entry is found, as much of the data in the entry as possible is copied to the passed buffer, and the passed length field is overlaid with the length of the data copied.

The following shows the function for reading a logical record cache entry:

```
/*          Read a cache entry               */

long      readCacheEntry(const cacheTokenPtr  cache_to_read,
                 const void     * primary_key,
                 const long     * primary_key_length,
                 const void     * secondary_key,
                 const long     * secondary_key_length,
                 const long     * size_of_buffer,
                 const void     * buffer);
```

If the entry is found, a CACHE_SUCCESS return code is returned to the caller. If no entry is found, a CACHE_NOT_FOUND return code is returned to the caller. When you receive this return code, you can retrieve the entry from permanent storage and add it to a logical record cache by using the `updateCacheEntry` function. See "Adding an Entry to a Logical Record Cache" on page 125 for more information about the `updateCacheEntry` function and using the function to add an entry to a logical record cache.

If the target cache was defined with a primary and secondary key, both a primary and a secondary key must be provided on each call to the `readCacheEntry` function. If the cache was defined with a primary key only, only a primary key should be passed. If a secondary key is provided, it is ignored by the `readCacheEntry` function.

Examples

The following example shows how to read an entry with primary and secondary keys from the processor shared cache that was created previously.

```
char primaryKey[ ] = "find the entry with this key";
char secondaryKey[ ] = "using this secondary key";
char buffer[255];
long primaryKeyL = 0;
long secondaryKeyL = 0;
long bufferL = 255;
```

```
primaryKeyL = strlen( primaryKey );        /* get length of primary Key          */
secondaryKeyL = strlen( secondaryKey );    /* get length of secondary Key        */

if( readCacheEntry (&myCacheShared,        /* addr of the token for the cache    */
                     primaryKey,           /* primary key                        */
                    &primaryKeyL,          /* primary key length                 */
                     secondaryKey,         /* secondary key                      */
                    &secondaryKeyL,        /* secondary key length               */
                    &bufferL,              /* length of buffer to return data in */
                     buffer )              /* address of buffer                  */
                      !=CACHE_SUCCESS)     /* successful read                    */
   {
      printf("entry not found reading Shared_Cache");   /* write message  */
      exit(1);                                          /* and exit       */
    }
```

The following example shows how to read an entry with a primary key from the processor unique cache that was created previously and has only primary keys.

```
strcpy( primaryKey,  "find the entry with this key");
primaryKeyL = strlen( primaryKey );            /* get length of primary Key */

if( readCacheEntry ( &myCacheUnique,        /* addr of the token for the cache    */
                      primaryKey,           /* primary key                        */
                     &primaryKeyL,          /* primary key length                 */
                      NULL,                 /* no secondary key                   */
                      NULL,                 /* no secondary key length            */
                     &bufferL,              /* length of buffer to return data in */
                      buffer )              /* address of buffer                  */
                       !=CACHE_SUCCESS)     /* successful read                    */
    {
      printf("entry not found reading Shared_Cache");   /* write message  */
      exit(1);                                          /* and exit       */
    }
```

Adding an Entry to a Logical Record Cache

To add an entry to a logical record cache, the application starts the `updateCacheEntry` function, passing the following as inputs:

- A pointer to the returned cacheToken value
- The primary and secondary key values and lengths
- A pointer to the data you want to add and the length of that data.

If the data is to be individually timed, you can pass a pointer to a timeout value. If not, set the timeout parameter to NULL. The `updateCacheEntry` function works as both an *add* when the entry is not already in cache or as an *update* when the entry already exists in cache. Therefore, the `updateCacheEntry` function can be used for both an add and an update; however, there is no way to determine whether the `updateCacheEntry` function performed an add or an update.

See *TPF C/C++ Language Support User’s Guide* for more information about the `updateCacheEntry` function. See “Updating an Entry in a Logical Record Cache” on page 127 for more information about updating an entry in a logical record cache.

The following shows the function to add an entry to a logical record cache:

```
long updateCacheEntry(const cacheTokenPtr cache_to_update,
             const void * primary_key,
             const long * primary_key_length,
             const void * secondary key,
             const long * secondary_key_length,
             const long * size_of_entry,
             const void * entry_data,
             const long * timeout,
             const char * invalidateOthers);
```

Primary and Secondary Keys

If the target cache was defined with both a primary and a secondary key, both a primary and secondary key must be specified on a call to the `updateCacheEntry` function. If the cache was defined with only a primary key, only a primary key is passed. If a secondary key is specified, the `updateCacheEntry` function ignores it.

invalidateOthers Parameter

The invalidateOthers parameter is used to identify the `updateCacheEntry` function as a change to local cache only or as a change to all caches in the complex. Because this call is used to add the entry to the local cache so that the next `readCacheEntry` will find the entry, set the invalidateOthers parameter to either a NULL pointer or a pointer to a char of `Cache_noInvalidate`.

timeout Parameter

The timeout parameter is used to associate a lifetime, in seconds, to the entry. If the entry remains in cache beyond this lifetime, the entry is invalidated and a CACHE_NOT_FOUND return code is returned to the next `readCacheEntry` function that tries to read the entry. The timeout parameter overrides the value specified for the castOutTime parameter on the `newCache` function call.

Examples

The following example shows how to add an entry with primary and secondary keys defined to the processor shared cache that was created previously.

```
char  primaryKey[ ] = "add the entry with this key";
char  secondaryKey[ ] = "using this secondary key";
char  data[255] = "This is the data entry";
long  primaryKeyL = 0;
long  secondaryKeyL = 0;
long  dataL = 0;
char  invalidateOption = Cache_NoInvalidate;

primaryKeyL = strlen( primaryKey );      /* get length of primary Key   */
secondaryKeyL = strlen( secondaryKey );  /* get length of secondary Key */
dataL = strlen( data );                  /* get length of data string   */


if( updateCacheEntry(
                    &myCacheShared,        /* addr of the token for the cache */
                     primaryKey,           /* primary key                     */
                    &primaryKeyL,          /* primary key length              */
                     secondaryKey,         /* secondary key                   */
                    &secondaryKeyL,        /* secondary key length            */
                    &dataL,                /* length of data to put in entry  */
                     data,                 /* address of data                 */
                     NULL,                 /* no timeout                      */
                    &invalidateOption)     /* invalidateOthers option         */


                      != CACHE_SUCCESS)    /* successful add                  */
    {
      printf("error on adding to Shared_Cache");   /* write message           */
      exit(1);                                     /* and exit                */
    }
```

The following example shows how to add an entry with a primary key to the processor unique cache that was created previously and has only primary keys defined.

```
 long       lifeTime = 180;               /*life time of 180 seconds (3 min) */

strcpy( primaryKey, "add the entry with this key");
 primaryKeyL = strlen( primaryKey );        /* get length of primary Key       */
 if( updateCacheEntry (
                      &myCacheUnique,       /* addr of the token for the cache */
                       primaryKey,          /* primary key                     */
                      &primaryKeyL,         /* primary key length              */
                       NULL,                /* no secondary key                */
                       NULL,                /* no secondary key length         */
                      &dataL,               /* length of data to put in entry  */
                       data,                /* address of data                 */
                      &lifeTime,            /* entry timeout                   */
                       NULL)                /* no invalidateOthers option      */
                       != CACHE_SUCCESS)    /* successful add                  */
    {
      printf("error adding entry to Unique_Cache");   /* write message  */
      exit(1);                                        /* and exit       */
    }
```

Updating an Entry in a Logical Record Cache

To update an entry in a logical record cache, the application starts the `updateCacheEntry` function, passing the following as inputs:

- A pointer to the returned cacheToken value
- The primary and secondary key values and lengths
- A pointer to the new data and the length of that data.

If the new data is to be individually timed, a pointer to a timeout value can be passed. If not, set the timeout parameter to NULL. The `updateCacheEntry` function works as both an *add* when the entry is not already in the cache or as an *update* when the entry already exists in the cache. Therefore, the `updateCacheEntry` function can be used for both an add and an update; however, there is no way to determine whether the `updateCacheEntry` function performed an add or an update.

See *TPF C/C++ Language Support User's Guide* for more information about the `updateCacheEntry` function. See "Adding an Entry to a Logical Record Cache" on page 125 for more information about adding an entry to a logical record cache.

The following shows the function to update an entry in a logical record cache:

```
long updateCacheEntry(const cacheTokenPtr cache_to_update,
             const void * primary key,
             const long * primary_key_length,
             const void * secondary key,
             const long * secondary_key_length,
             const long * size_of_entry,
             const void * entry_data,
             const long * timeout,
             const char * invalidateOthers);
```

Primary and Secondary Keys

If the target cache was defined with both a primary and a secondary key, both a primary and a secondary key must be provided on each call to the `updateCacheEntry` function. If the cache was defined with a primary key only, only a primary key is passed. If a secondary key is specified, the `updateCacheEntry` function ignores it.

invalidateOthers Parameter

The invalidateOthers parameter is used to identify this `updateCacheEntry` as a change to the local cache only (an *add* call; see “Adding an Entry to a Logical Record Cache” on page 125 for more information about adding an entry to a logical record cache) or as a change to all caches in the complex (an *update* call). If the cache is a processor shared cache, the invalidateOthers parameter is then used to tell the logical record cache to invalidate this entry in all processors. If the entry is marked as an *update* type by setting the invalidateOthers value to `Cache_Invalidate`, the logical record cache will start CF support to inform any other processor that has registered interest in this cache entry that it has been changed. Do this when the source information is actually changed on permanent storage and is not used if this call was just to add the information to the local cache. Using the invalidateOthers parameter the wrong way can cause performance problems. If the invalidateOthers parameter is set to either a NULL pointer or a pointer to a char of `Cache_NoInvalidate`, the `updateCacheEntry` function call is considered an add or update of the entry in the local cache only. If the cache is a processor unique cache, there is no difference between the two types of calls (add or update) and the invalidateOthers parameter is ignored.

Note: When a processor shared cache is running in *local* mode because CF support has not been installed, the value of the invalidateOthers parameter has no effect.

timeout Parameter

The timeout parameter is used to associate a lifetime, in seconds, to the entry. If the entry remains in the cache beyond this lifetime, the entry is invalidated and a CACHE_NOT_ FOUND return code is returned to the next `readCacheEntry` function that tries to read the entry. The new timeout parameter overrides the value specified for the castOutTime parameter on the `newCache` function call, overlays the value the updated entry may contain, and restarts the timeout function for the entry using the new value.

Examples

The following example shows how to update an entry with primary and secondary keys in the processor shared cache that was created previously and how to cause any other processors with the same entry to have the entry marked as invalidated.

```
char  primaryKey[ ] = "update the entry with this key";
char  secondaryKey[ ] = "using this secondary key";
char  data[255] = "This is the data entry";
long  primaryKeyL = 0;
long  secondaryKeyL = 0;
long  dataL = 0;
char  invalidateOption = Cache_Invalidate;  /*force invalidation of other processors */

primaryKeyL = strlen( primaryKey );      /* get length of primary Key   */
secondaryKeyL = strlen( secondaryKey );  /* get length of secondary Key */
dataL = strlen( data );                  /* get length of data string   */

if( updateCacheEntry (
                    &myCacheUnique,       /* addr of the token for the cache */
                     primaryKey,          /* primary key                     */
                    &primaryKeyL,         /* primary key length              */
                     NULL,                /* no secondary key                */
                     NULL,                /* no secondary key length         */
                    &dataL,               /* length of data to put in entry  */
                     data,                /* address of data                 */
                    &lifeTime,            /* entry timeout                   */
                    &invalidateOption)    /* invalidateOthers option         */
```

```
                 != CACHE_SUCCESS)    /* successful add                */
{
  printf("error updating entry in Unique_Cache");  /* write message  */
  exit(1);                                         /* and exit       */
}
```

Deleting an Entry from a Logical Record Cache

To delete an entry from a logical record cache, the application starts the `deleteCacheEntry` function, passing the following as inputs:

- A pointer to the returned cacheToken value
- The primary and secondary key values.

The primary and secondary keys must exactly match the primary and secondary keys that are used to add the entry to cache. This includes both the content of the keys and their lengths. Additionally, the ECB must have the same DBI as the ECB that is used to add the entry. See “Adding an Entry to a Logical Record Cache” on page 125 for more information about adding an entry to a logical record cache.

See *TPF C/C++ Language Support User’s Guide* for more information about the `deleteCacheEntry` function.

The following shows the function to delete an entry from a logical record cache:

```
long  deleteCacheEntry(const cacheTokenPtr cache_to_update),
              const void  * primary_key,
              const long  * primary_key_length,
              const void  * secondary_key,
              const long  * secondary_key_length);
```

If the target cache was defined with a primary and a secondary key, both a primary and secondary key must be provided on each call to the `deleteCacheEntry` function. If the cache was defined with a primary key only, then only a primary key should be passed. If a secondary key is provided, it is ignored by the `deleteCacheEntry` function.

If the target cache is a processor shared cache using the CF, the `deleteCacheEntry` function causes all other processors to have their copies of the deleted entry invalidated.

Examples

The following example shows how to delete an entry with primary and secondary keys defined in the processor shared cache that was created previously and how to cause any other processors with the same entry to have that entry marked as invalidated.

```
char  primaryKey[ ] = "delete the entry with this key";
char  secondaryKey[ ] = "using this secondary key";
long  primaryKeyL = 0;
long  secondaryKeyL = 0;

primaryKeyL = strlen( primaryKey );       /* get length of primary Key    */
secondaryKeyL = strlen( secondaryKey );   /* get length of secondary Key  */

if( deleteCacheEntry (
                    &myCacheShared,      /* addr of the token for the cache */
                     primaryKey,         /* primary key                     */
                    &primaryKeyL,        /* primary key length              */
                     secondaryKey,       /* secondary key                   */
                    &secondaryKeyL)      /* secondary key length            */
```

```
                      != CACHE_SUCCESS)  /* successful delete               */
    {
      printf("error deleting entry in Shared_Cache");  /* write message  */
      exit(1);                                         /* and exit       */
    }
```

The following example shows how to delete an entry with a primary key defined in a processor unique cache that was previously created with only primary keys.

```
strcpy( primaryKey,   "delete the entry with this key");
 primaryKeyL = strlen( primaryKey );              /* get length of primary Key*/

 if( deleteCacheEntry (
                    &myCacheUnique,       /* addr of the token for the cache */
                     primaryKey,          /* primary key                     */
                    &primaryKeyL,         /* primary key length              */
                     NULL,                /* no secondary key                */
                     NULL)                /* no secondary key length         */
                     !=CACHE_SUCCESS)     /* successful delete               */
     {
      printf("error deleting entry in Unique_Cache");  /* write message  */
      exit(1);                                         /* and exit       */
    }
```

Flushing Entries from a Logical Record Cache

To flush all the entries from a logical record cache, the application starts the `flushCache` function, passing a pointer to the returned cacheToken value for the cache to be flushed as input. The flush occurs immediately although other applications may also be using the cache. If the cache is a processor shared cache, no notification is sent to any other processors and no entries in their copy of the cache are affected.

See *TPF C/C++ Language Support User's Guide* for more information about the `flushCache` function.

The following example shows the function to flush a logical record cache:

```
long   flushCache( cacheTokenPtr cache_to_flush);
```

Examples

The following example shows how to flush all entries from the processor shared cache that was created previously.

```
flushCache ( &myCacheShared );    /* addr of the token for the cache  */

printf(" Shared_Cache flushed");  /* write message                    */

exit(0);                          /* and exit                         */
```

The following shows how to flush all entries from the processor unique cache that was created previously.

```
flushCache ( &myCacheUnique );    /* addr of the token for the cache  */

printf(" Unique_Cache flushed");  /* write message                    */

exit(0);                          /* and exit                         */
```

Deleting a Logical Record Cache

To delete a logical record cache, the application starts the `deleteCache` function, passing a pointer to the returned cacheToken value for the cache to be deleted as input. This delete occurs immediately although other applications may also be using the cache. Users of the cache will receive a CACHE_ERROR_TOKEN return code when they try to access the deleted cache.

If the cache is a processor shared cache, the connection to the CF cache structure is deleted. If this is the last processor using the CF cache structure in the CF, the structure is deleted. Otherwise, no other notification is sent to any other processor and no entries in their copy of the cache are affected. See *TPF Database Reference* for more information about CF cache structures.

See *TPF C/C++ Language Support User's Guide* for more information about the `deleteCache` function.

The following shows the function to delete a logical record cache:

```
long  deleteCache( cacheTokenPtr cache_to_delete);
```

Examples

The following example shows how to delete a processor shared cache that was created previously.

```
deleteCache ( &myCacheShared );   /* addr of the token for the cache  */

printf(" Shared_Cache deleted");  /* write message                    */
exit(0);                          /* and exit                         */
```

The following example shows how to a delete processor unique cache that was created previously.

```
deleteCache ( &myCacheUnique );   /* addr of the token for the cache */

printf("Unique_Cache deleted");   /* write message                   */

exit(0);                          /* and exit                        */
```

Writing TPF Application Programs in C and C++

This chapter teaches you what you need to know to write TPF programs in C and C++ languages. It contains a limited amount of information about C and C++ languages and compilers.

For detailed information about the many TPF application programming interface (API) functions that are referred to in this chapter (including how to code them), see the *TPF C/C++ Language Support User's Guide*. For information about the functions that are provided with TCP/IP support, see *TPF Transmission Control Protocol/Internet Protocol*.

This chapter contains the following sections:

- Special TPF Considerations discusses characteristics of the TPF operating system that affect programming in C and C++.
- TPF Header Files describes header files included for the TPF operating system, their relationship to each other, and how to create your own.
- TPF Application Environment tells you how to perform specific TPF tasks using C and C++ language support.
- Calling Other Functions and Programs tells you how function linkage is performed and how parameters are passed.
- Compiling and Running C/C++ Programs describes TPF-related C and C++ compiler options and how using them affects your compiled programs.
- DLL Compiler Option tells you when to use this compiler option and what it means to use the DLL compiler option.

Special TPF Considerations

There are some characteristics of the TPF operating system that will affect your programming.

Coding `main` Functions

The TPF system supports coding `main` functions in dynamic load modules (DLMs) and passing the standard C `argc` and `argv` parameters to them. A DLM can contain, at most, one `main` function. If a DLM does contain a `main` function, the main function is the DLM entry point.

Building and Loading DLMs Containing a `main` Function

There is no change to the DLM build or load process. DLMs that contain a `main` function are compiled, prelinked, link-edited, and loaded exactly the same way as DLMs have been before this support.

The TPF offline loader (TPFLDR) program, the DLM startup code (segment CSTRTD, which must be linked into every DLM, including those that contain a `main` function), and the TPF C run-time environment initialization code (segment CLMINT in the CISO library) detect the presence of the `main` function and manage it appropriately.

CLMINT also includes a function similar to a UNIX shell that parses command strings passed with the `system` function into `argc` and `argv` parameters for the `main` function. This same function will also parse a command string contained in a core block attached to data level 0 (D0) of the ECB that is passed by various ECB create functions.

Defining a `main` Function

The `main` function can be defined anywhere in a DLM. The definition of `main` should return an `int`, and take either no parameters:

```
int main(void) { /* code for main */ }
```

or two parameters:

- `int`, which is set to the number of argument strings passed to `main`
- `char**`, which is set to the vector of pointers to argument strings passed to `main`.

These parameters are conventionally named `argc` and `argv`; for example:

```
int main(int argc, char **argv) { /* code for main */ }
```

The constraints on these variables are as follows:

- `argc` >= 0.
- `argv[argc]` is a null pointer.
- `argv[0]` through `argv[argc-1]` point to NUL (`'\0'`) terminated argument strings.
- if `argc` >= 0, `argv[0]` contains the load module name.

Calling a DLM Containing a `main` Function

The TPF system supports calling a DLM containing a `main` function through the `system` function which is a standard C library function. The `system` function creates a new ECB, which runs synchronously while the calling ECB waits. In UNIX terms, the ECB that calls the `system` function is the *parent process*; the created ECB is the *child process*. When the child process exits, its exit value is returned to the parent process as the `system` function's return value, and the parent process resumes running.

In addition, the TPF system supports calling a DLM containing a `main` function through several TPF-unique ECB create functions or macros. The supported functions and macros include:

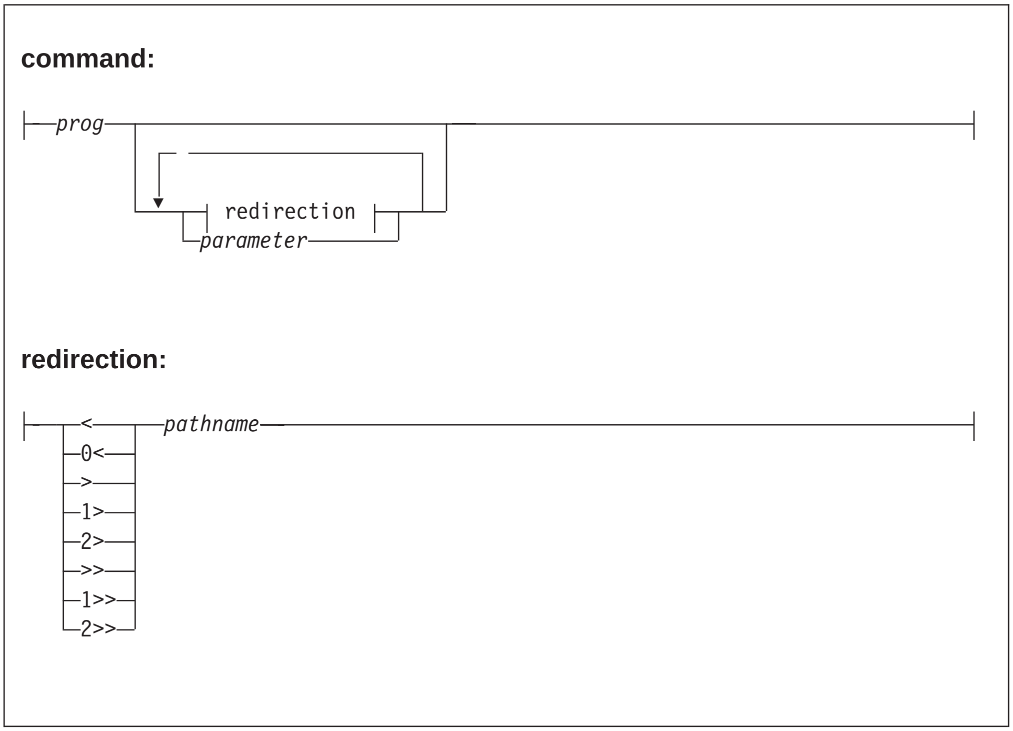
- `creec` function
- `cretc_level` function
- `swisc_create` function
- `tpf_cresc` function
- CREEC macro
- CRESC macro
- CRETC macro
- CXFRC macro
- SWISC macro.

Passing Arguments to `main`

When a process calls the `system` function and creates a child process, the TPF system parses the `system` command string into `argc` and `argv` parameters for the `main` function of the child process.

When a process calls one of the TPF-unique functions that will enter a DLM, the TPF system checks data level 0 (D0) of the ECB for a core block that contains the command string. If D0 is unoccupied, `argc` is set to 0 and `argv[0]` is set to the program name. However, if there is a core block at D0 of the ECB, the TPF system parses the command string, starting at byte 0 of the core block, into `argc` and `argv` parameters for the `main` function. The TPF system then releases the core block.

The TPF system uses the following format to parse command strings:



Where

command
A string that creates a new process, giving the program name and any additional parameters and standard stream redirections associated with the new process.

prog
The name of the TPF program segment that contains the `main` function.

redirection
Specification of a file that is to be opened as one of the standard streams.

< or `0<`
Indicates redirection of the `stdin` stream.

> or `1>`
Indicates redirection of the `stdout` stream. If the file exists, it is truncated to zero bytes.

`2>` Indicates redirection of the `stderr` stream. If the file exists, it is truncated to zero bytes.

>> or `1>>`
Indicates redirection of the `stdout` stream without truncation and with output appended to the end of the file.

`2>>`
Indicates redirection of the `stderr` stream without truncation and with output appended to the end of the file.

pathname
The path to the file from or to which the stream will be redirected.

parameter
A blank delimited substring of the command string that contains no blank characters and cannot be parsed as a redirection. Each *parameter* is passed as an element of the `argv` vector to the `main` function in program *name*.

The TPF system:

- Formats the command string into the `argv` vector so that:
  - `argv[0]` contains the address of the *prog* string.
  - `argv[1]` to `argv[`*n*`]` contains the addresses of any *parameter* strings.
  - `argv[`*n*`+1]` contains a null pointer.
- Builds a parameter list. This parameter list contains the following two parameters:
  - An `int` parameter, which contains the total number of strings in the `argv` array (*n*+1)
  - The address of the `argv` array.
- Sets the current working directory as specified by the TPF_CWD_PATHNAME environment variable, or to the root directory if TPF_CWD_PATHNAME does not specify a directory.
- Opens the standard streams on the files specified by:
  1. The redirection specifications in the command string
  2. If there is no redirection specification for the standard stream, it is opened on the path specified by environment variable TPF_STDIN_PATHNAME, TPF_STDOUT_PATHNAME, or TPF_STDERR_PATHNAME
  3. If no path name is specified for the standard stream, either by a redirection specification or by an environment variable, the standard stream is not opened and any I/O operations on that stream will fail unless a call to the `freopen` function successfully opens the standard stream.

I/O Stream Pipes

Many C run-time environments support piping the standard output (`stdout`) stream from one process to the standard input (`stdin`) stream of a second process. The TPF system does not accept command strings that contain vertical bars (|) (which are used by UNIX and other C run-time environments to indicate I/O stream pipes) when you are using the `system` function. If a string containing a vertical bar is passed to the `system` function, the `system` function returns −1 and sets `errno` to `EINVAL`. For more information about I/O stream pipes, see the information for the `mkfifo`, `tpf_fork`, and `pipe` functions in *TPF C/C++ Language Support User's Guide*.

Example of Calling a DLM That Contains a `main` Function

If the following program ABCD:

```
/* Demonstration program ABCD */

#include <stdio.h>
#define _POSIX_SOURCE
#include <stdlib.h>

int main(int argc, char **argv)
{
    FILE *fp = fopen("wxyz.input", "w");
    fputs("A message from ABCD.\n", fp);
    fputs("TPF now supports:\n\n", fp);
    fputs("--  int main(void);\n", fp);
    fputs("--  int main(int argc, char ** argv);\n", fp);
    fputs("--  environment variables, inherited through system();\n",
          fp);
    fputs("--  standard I/O stream redirection.\n", fp);
    fclose(fp);

    setenv("myname", argv[0], 1);
    printf("%s:  My name is %s.\n", argv[0], getenv("myname"));

    printf("%s:  Now I will execute program WXYZ.\n", argv[0]);

    printf("%s:  WXYZ returned %d\n", argv[0],
           system("WXYZ <wxyz.input one two three"));

    printf("%s:  My name is still %s\n", argv[0], getenv("myname"));
    remove("wxyz.input");

    return 0;
}
```

invokes the following program WXYZ:

```
/* Demonstration program WXYZ */

#include <stdio.h>
#include <limits.h>
#define _POSIX_SOURCE
#include <stdlib.h>

int main(int argc, char **argv)
{
    int i;
    char buffer[256];

    printf("%s:  My parent's name is %s\n", argv[0], getenv("myname"));
    setenv("myname", argv[0], 1);
    printf("%s:  My name is %s\n", argv[0], getenv("myname"));

    printf("%s:  argc = %d\n", argv[0], argc);
    for (i = 0; i < argc; ++i)
    {
        printf("%s:  argv[%d] = %s\n", argv[0], i, &cond.
argv[i]);
    }

    for (i = 1; gets(buffer); ++i)
    {
        printf("%s:  Line %d of stdin:  %s\n", argv[0], i, buffer);
    }

    return 42;
}
```

The output will then be:

```
ABCD:  My name is ABCD.
ABCD:  Now I will execute program WXYZ.
WXYZ:  My parent's name is ABCD
WXYZ:  My name is WXYZ
WXYZ:  argc = 4
WXYZ:  argv[0] = WXYZ
WXYZ:  argv[1] = one
WXYZ:  argv[2] = two
WXYZ:  argv[3] = three
WXYZ:  Line 1 of stdin:  A message from ABCD.
WXYZ:  Line 2 of stdin:  TPF now supports:
WXYZ:  Line 3 of stdin:
WXYZ:  Line 4 of stdin:  --  int main(void);
WXYZ:  Line 5 of stdin:  --  int main(int argc, char ** argv);
```

```
WXYZ:  Line 6 of stdin:  --  environment variables, inherited through system();
WXYZ:  Line 7 of stdin:  --  standard I/O stream redirection.
ABCD:  WXYZ returned 42
ABCD:  My name is still ABCD
```

Note: In the previous example, program WXYZ inherits a copy of the ABCD environment list, but changes that WXYZ makes to its environment list are not copied back to the ABCD environment list.

TARGET(TPF) Restriction

While we are talking about functions, there is something else that you should keep in mind. If a function linkage is not explicitly specified (coded without `static` or `extern` specified), the default is `extern`. Although there is no requirement to explicitly type functions, it is recommended for TPF systems.

If you change the declaration of a function from `static` to `extern`, you must change the allocator table so that external linkage is provided for the new entry point:

1. Code a statement in SIP skeleton IBMPAL for each new external function. Use the format for a transfer vector as described in *TPF System Installation Support Reference*.
2. Create a new version of the allocator.

Example of Creating an ECB That Enters a DLM That Contains a `main` Function

If the following program EFGH:

```
#include <tpfapi.h>
#include <stdlib.h>
#include <stdio.h>

#define PROG_NAME      "EFGH"
#define COMMAND_STRING "STUV Parm#1 Parm#2 TESTParm"

void EFGH (void)
{
  int seconds;      /* CRETC time interval         */
  char *coreBlock;  /* pointer to core block       */
  char *action;     /* action word passed by CRETC */

  printf ("%s: Creating ECB to enter DLM main.\n",
           PROG_NAME);
  /****************************************************/
  /* Release data level 3, if held. Then get a new     */
  /* core block which will hold command string for     */
  /* the main function.                                */
  /****************************************************/
  crusa (1, D3);
  coreBlock = (char *)getcc(D3, GETCC_TYPE, L2);

  memset (coreBlock, 0x00, ecbptr()->ce1cc3);

  /****************************************************/
  /* Setup the command string for the main function.   */
  /****************************************************/
  strcpy (coreBlock,
          COMMAND_STRING,
          strlen(COMMAND_STRING));
  /*****************************************************/
  /* Create a time-initiated ECB.                      */
  /*****************************************************/
  seconds = 10;
  action = "TPF0";
  cretc_level (CRETC_SECONDS, STUV, seconds, action, D3);

  /*****************************************************/
  /* Print completion message.                         */
  /*****************************************************/
  printf ("%s:  ECB will be created in %d seconds.\n",
            PROG_NAME, seconds);

  exit(0);

}
```

invokes the following program STUV:

```
#include <stdlib.h>
#include <stdio.h>

#define PROG_NAME "STUV"

int main (int argc, char **argv)
{
  int i;           /* Loop counter */
  char *action;    /* Action code passed by caller */

  printf ("%s:  New ECB created successfully.\n",
            PROG_NAME);

  action = (char *)&ecbptr;()->ebw000;
  action[4] = '\0';

  /*****************************************************/
  /* Print the action code which was passed by caller. */
  /*****************************************************/
  printf ("%s:  Action code = %s.\n", PROG_NAME, action);

  /*****************************************************/
  /* Print the name of our program and then loop       */
  /* through each of the parameters passed in the argv */
  /* parameter.                                        */
  /*****************************************************/
  printf ("%s:  Our program name is %s.\n", PROG_NAME,
            argv[0]);

  for (i = 1; i < argc; i++)
  }
       printf ("%s:  Parameter #%d is %s.\n", PROG_NAME,
                 i, argv[i]);
  }

  /*****************************************************/
  /* Exit this ECB.                                    */
  /*****************************************************/
  printf ("%s:  Created ECB is now exiting.\n",
            PROG_NAME);

  exit (0);

}
```

The output will then be:

```
EFGH:  Creating ECB to enter DLM main.
EFGH:  ECB will be created in 10 seconds.
STUV:  New ECB created successfully.
STUV:  Action code = TPF0.
STUV:  Our program name is STUV.
STUV:  Parameter #1 is Parm#1.
STUV:  Parameter #2 is Parm#2.
STUV:  Parameter #3 is TESTParm.
STUV:  Created ECB is now exiting.
```

Coding C++ Applications

This section describes some of the concepts you need to know when coding C++ applications. See the IBM C/C++ user's guide and programmer's guide for the System/390 platform used by your installation for a more thorough discussion about C++ and dynamic link libraries (DLLs).

`main` or `extern "C"` Requirement

DLL applications are required to have either `main` or an entry point with the same name as the load module. If your DLL application does not have `main`, C++ *mangles* the function name. You code an `extern "C"` linkage specification to produce an entry point with the same 4-character uppercase name as the load module.

The `extern "C"` linkage specification allows a C++ application entry point to be called through TPF enter/back services. This linkage specification also forces the linkage to the entry point of the load module to be C linkage instead of C++ linkage.

Only the entry point function must have the `extern "C"` linkage specification. Other functions in a C++ application do not need this linkage specification. A function in the C++ application that is called by another function in the same load module can have C++ linkage.

If you do not code the `extern "C"` linkage specification, the offline loader (TPFLDR) provides an error message that the entry point is not found in the program.

In the following example, the `extern "C"` linkage specification produces an entry point with the same name as the QZZ0 load module name. The call to `ReadIt` in `EmpClass` does not require this linkage specification.

```
class EmpClass
{
...
}

extern "C" void QZZ0 ();
{
  double raise;
  EmpClass *EmpPtr = new EmpClass[total_employees];
...
  raise = EmpPtr[i].ReadIt(raise);
...
}

double & EmpClass::ReadIt (double & rate)
{
...
...
}
```

C++ Exceptions

This section describes some of the TPF system considerations you need to know when coding C++ exceptions in your application. See the *OS/390 C/C++ Language Reference* for more information about general C++ exception handling.

The TPF system supports only application-defined exceptions. System errors and program checks are handled by the operating system and are not surfaced to the application.

Exceptions can be thrown across load module boundaries; for example, an exception can be thrown by a DLM but caught by another DLM that resides in the ECB program nesting level (PNL). All TPF programs that reside in the PNL between the current program and the program that contains a catch clause are dropped from the nesting level. TPF enter/back processing performs all clean up that is associated with dropping a program from a nesting level.

Note: Do not use C++ exceptions in applications that have parts written in (TARGET)TPF.

If a TARGET(TPF) program is dropped from a nesting level during exception processing, a SNAP system error is taken and the application continues to run; however, application problems may occur if the application calls a TARGET(TPF) program after the exception is processed.

If an exception is thrown, but a catch clause in the application cannot be found to handle the exception, the standard behavior is that the `terminate` function is called. The default action of the `terminate` function is to call the `abort` function, which raises the SIGABRT signal; however, the `terminate` function is modified to produce a system error and exit the ECB instead of calling the `abort` function.

Note: For this reason, do not write applications to use both signals and exceptions.

If you code a `throw` block in a destructor, the corresponding `try` and `catch` blocks must also be in the scope of the destructor or the results are unpredictable.

Exporting

Exporting is a DLL concept. The following are ways to export functions and variables:

- The EXPORTALL compiler option allows a DLL to export all external functions and variables. This also means that no functions or variables can be hidden; everything in the DLL is accessible to other DLLs and DLL applications. If you use EXPORTALL, you do not need to include the `#pragma export` directive.
- The `#pragma export` directive permits you to control specific functions and functions that can be exported when you can code this directive in your source file.
- Using the EXPORT C++ language extension keyword permits you to declare that the function or variable is to be exported.

Reentrant Programming

All TPF application programs must be reentrant. This means that a TPF application program must be coded as if it was being run simultaneously by more than one process. This is called *parallel reentrancy*. Furthermore, TPF applications must not leave anything behind that alters the path of a subsequent process. This is called *serial reentrancy*.

TPF systems provides 2 ways of defining data objects to be accessed by more than 1 program (globals).

- For data that remains across multiple ECBs, use globals. For more information about globals, see “Using TPF Globals” on page 155.
- For data that does not remain across ECBs, the ECB work area and data levels can be used.

Addresses can be passed between functions in the same DLM. Passing function pointers between DLMs can result in errors, because in most cases the static storage is not set up properly.

Note: DLL and C++ applications can pass function pointers between themselves.

Programming Rule

Pass addresses between functions in the same C program. Do not pass function pointers between C programs.

It is possible, though difficult, to write self-modifying code in C. Because of reentrancy requirements, this practice is not allowed in TPF systems.

Standard TPF Program Sizes

ISO-C programs can exceed 4095 bytes (4KB). The load module representing an ISO-C program can be the largest size supported by the linkage editor. The load module is stored on online DASD in 4KB chained records. When loaded into main storage, contiguous storage is used.

TARGET(TPF) Restriction

TARGET(TPF) programs are restricted to 4KB blocks. If you need to break up an existing C source program into smaller segments, do the following:

- Ensure that the appropriate set of header files is duplicated in the new segments.
- Ensure that any required function prototypes and structure declarations are present in the new segments. Prototypes for static functions must be modified to reflect the fact that they now reside in a different source module and are, therefore, external.
- There can be certain static functions that are called by both the functions being placed in the new segment, and the functions that remain in the old; these must be duplicated in both segments.

```
#include <stdlib.h>
#include <stdio.h>

#define PROG_NAME   "STUV"

int main (int argc, char **argv)
{
  int i;          /* Loop counter */
  char *action;   /* Action code passed by caller */

  printf ("%s:  New ECB created successfully.\n",
            PROG_NAME);

  action = (char *)ecbptr()->ebw000;
```

```
   /****************************************************/
   /* Print the action code which was passed by caller. */
   /****************************************************/
   printf ("%s:  Action code = %s.\n", PROG_NAME, action);

   /****************************************************/
   /* Print the name of our program and then loop      */
   /* through each of the parameters passed in the argv */
   /* parameter.                                       */
   /****************************************************/
   printf ("%s:  Our program name is %s.\n", PROG_NAME,
             argv[0]);

   for (i = 1; i < argc; i++)
   {
        printf ("%s:  Parameter #%d is %s.\n", PROG_NAME,
                  i, argv[i]);
   }

   /****************************************************/
   /* Exit this ECB.                                   */
   /****************************************************/
   printf ("%s:  Created ECB is now exiting.\n",
             PROG_NAME);

   exit (0);

}
```

For planning purposes, it is a good idea to map out all of the functions in the original source module and determine which functions are called by other functions.

Static Storage Considerations

As described in Understanding High-Level Language Concepts in the TPF System, static storage is the storage required for static variables needed by a given ECB. Once acquired, this storage is not released until the ECB exits. (Static storage is not released with an ENTDC for ISO-C but it is released for TARGET(TPF).)

TPF Header Files

Several header files are provided for TPF application programming. The header files perform the following functions:

- Map the ECB, IBM-released globals, and other TPF data structures
- Provide C function prototypes
- Provide constant definitions for the function parameters and return codes
- Define macros for the functions implemented as macros
- Define the `TPF_regs` structure (`TPF_regs` is a C structure used to pass parameter values to assembly language programs in registers).

The header file structure approach changed from TARGET(TPF) to be more aligned with the way standard C programs are written. TARGET(TPF) provides the following for the `tpfeq.h` header file:

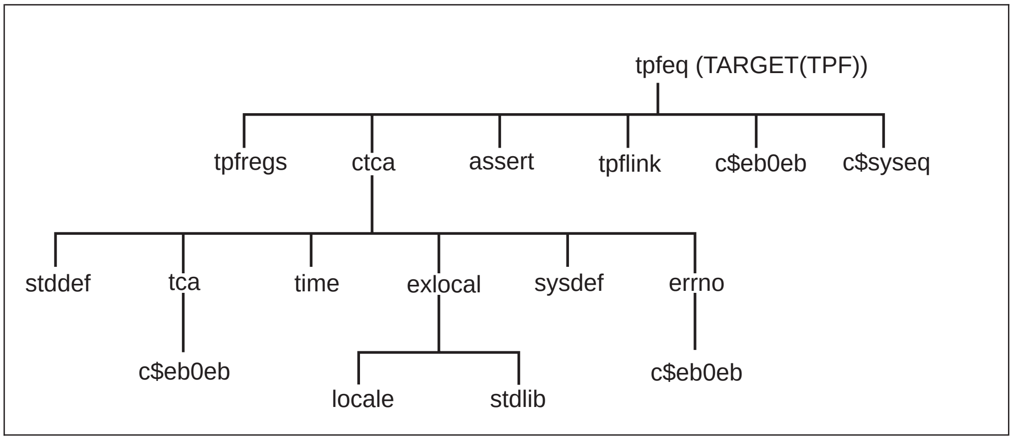


Figure 17. `tpfeq.h` Header File for TARGET(TPF)

ISO-C provides the following for the `tpfeq.h` header file:

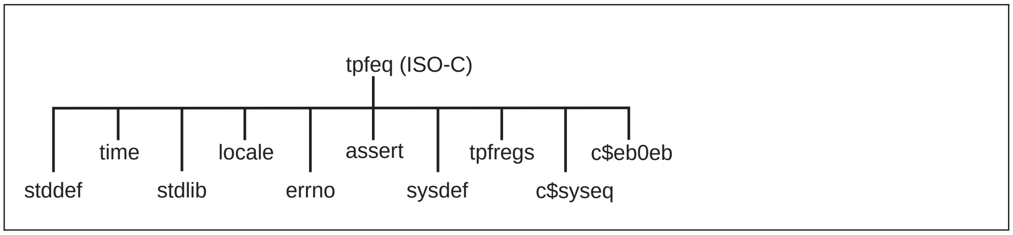


Figure 18. `tpfeq.h` Header File for ISO-C

Creating Your Own Header Files

In addition to the header files that the TPF system provides, you can create your own. This section gives you some general guidelines.

The following items belong in TPF header files:

- Commonly defined constant terms using the `#define` preprocessor statement. These include:
  - TPF constants
  - Bit masks used to test for particular conditions
  - Other uses of the `#define` statement, such as coding a `#define` statement for a constant following an `#ifndef` statement, to make sure the constant is not defined more than once.
- `struct`, `enum`, and `union` declarations (***not definitions***, which actually reserve space for the data objects)
- C macros
- Entry point prototypes

  Note: It is recommended that you pass structures to a function by reference and not by value. If the function does not update the structure, pass a pointer to a `constant` structure.
- Type definitions (using the `typedef` facility)
- Any number of nested `#include` statements.

The following items do ***not*** belong in TPF header files:

- Data object definitions of any type or class

- Function code of any type.

All data structures that have a corresponding assembler DSECT defined by their own macro reside in their own header files. If you are going to convert existing assembler data macros to C or C++ structures, you need to consider boundary alignment. Boundary alignment is determined by how data types are stored. The following shows how data types are stored and aligned by the C or C++ compiler:

Data Type	**Memory Occupied**	**Alignment**
`char`	1 byte	byte
`int`	4 bytes	fullword
`short int`	2 bytes	halfword
`long int`	4 bytes	fullword
`float`	4 bytes	fullword
`double`	8 bytes	doubleword

In C, using `int`, `short int`, `long int`, `float`, and `double` in data structures forces boundary alignment to the next boundary determined by the data type. When these data structures are coded in assembler, the boundaries can differ (unless the assembler version is coded to coincide with the C version). This is a serious problem because it can result in errors that are difficult to diagnose.

TPF Header Files and C++

The header file structure of C programs is changed to handle C++. The following changes have been made:

1. The header file prolog will look as follows:

   ```
   ??=ifndef __NAME_HEADER__
   ??=ifdef __COMPILER_VER__
   ??=pragma filetag("IBM-1047")
   ??=endif
   #define __NAME_HEADER__  1
   ```

   See *TPF Programming Standards* for an example of the C header prolog.
2. The `#pragma margins` directive is deleted from the code. The following example shows how the `#pragma margins` directive was previously used.

   ```
   ??=pragma margins(1,72) sequence(73,80)
   ```
3. C linkage wrappers are added around all function prototypes, so all the existing C functions can be called from the C++ programs.

   ```
   #ifdef __cplusplus
       extern "C" {
   #endif

   /* C function prototypes */

   #ifdef __cplusplus
       }
   #endif
   ```
4. The definition of NULL for C++ is added as follows:

   ```
   #ifdef __cplusplus
       #define NULL  0
   #else
       #define NULL  ((void *) 0)
   #endif
   ```

5. The `#pragma pack` directive is supported for C++. The `#pragma pack` directive specifies the alignment roles to use for the structures, unions, and classes. The C++ compiler does not support the `_Packed` qualifier on the structure declaration. See the following example:

   ```
   #ifdef __cplusplus
      #pragma pack (packed)
   #else
   _Packed
   #endif
   struct
   {
     long  int li1;
     char      cc1;
     short int si1;
     long  int li2;
     short int si2;
     char      cc2;
   } mytest;

   #ifdef __cplusplus
      #pragma pack (reset)
   #endif
   ```

   See the IBM C/C++ language reference on the System/390 platform used by your installation for more information about `#pragma pack`.
6. The `#pragma linkage` directive is not supported in C++ language. Therefore, change the `#pragma linkage` (func_name, builtin) directive in C to read:

   ```
   #ifdef __cplusplus
     extern "builtin"
   #else
     #pragma linkage(func_name,builtin)
   ```

More Useful Information

The following items can be useful.

1. If you need to create a C structure from an existing BAL DSECT, use the CDSECT tool. The tool is documented in the user's guide for the particular IBM C or C++ compiler used by your installation. See "IBM High-Level Language Books" on page xix for a list of IBM C and C++ compiler publications for the System/390 platform. Using the CDSECT tool will aid in avoiding code language-porting errors.

   When you run the CDSECT tool, these are the recommended parameters:

   ```
   CDSECT name(BITF0XL DEFSUB EQUATE(DEF) LOWERCASE SEQUENCE)
   ```
2. Indicate all storage generated for boundary alignment in a global/external structure by using a bit declaration and a comment.

   ```
   struct tpf_example {
                      long  int exam_long;  /* Field one              */
                      short int exam_short; /* Field two              */
                            int : 16;       /* Reserved for use by IBM */
                      long  int exam_long1; /* Field three            */
                     };
   ```
3. Declare all reserved storage in a structure by using an unnamed bit declaration and a comment.

   Use no more than 16-bit increments to declare unnamed bit fields. If more than 2 bytes (16 bits) are needed, then use as many instances of 16-bit declarations as necessary.

   ```
   struct tpf_example { long  int  exam_long;       /* Field one              */
                              int        : 16;      /* Reserved for use by IBM */
                              int        : 16;      /* Reserved for use by IBM */
   ```

```
                     long  int  exam_long1;      /* Field two              */
                   };
struct tpf_example2 { long  int  exam_long;      /* Field one              */
                            char exam_long1;     /* Field two              */
                            int       : 16;      /* Reserved for use by IBM */
                            int       : 8;       /* Reserved for use by IBM */};
```

4. Order the fields within a structure according to the basic integral boundary alignment (double, fullword, halfword, character) when creating new structures. This avoids boundary alignment problems which can cause code errors.

   To take advantage of field characteristics with C addressability, here is a suggested use of pattern:

```
struct tpf_example { long  int  exam_long;       /* long example           */
                     short int  exam_short;      /* short example          */
                     short int  exam_short2;     /* 2nd short example      */
                     short int  exam_short3;     /* 3rd short example      */
                           char exam_char;       /* character example      */
                           char exam_char2;      /* 2nd character example  */ };
```

   instead of:

```
struct tpf_example {       char  exam_char;      /* character example      */
                           int   : 8;            /* Reserved for use by IBM */
                     short int  exam_short;      /* short example          */
                     short int  exam_short2;     /* 2nd short example      */
                     short int  exam_short3;     /* 3rd short example      */
                           char exam_char2;      /* 2nd character example  */
                           int  : 16;            /* Reserved for use by IBM */
                           int  : 8;             /* Reserved for use by IBM */
                     long  int  exam_long;       /* long example           */ }
```

   Even if you need to organize a structure by function, this convention still applies. A substructure is created to contain the function, and the fields in the substructure are ordered by their integral boundary alignment.

Programming Rule

When data is used by both C or C++ language and assembler language programs, define the storage consistently with C or C++ language and adapt the assembler language programs accordingly.

When an `int` field is defined as a bit field, the C compiler does not align the field, unless there is an intervening 0-length field.

Example: Assume you have 3 bytes of characters, followed by an integer. In assembler, it might be coded as follows:

```
CHARS    DS     CL3
INTEGER  DS     XL4
```

Assuming the character variable started at displacement 0, the integer variable will start in location 3.

Assume the same structure is coded in C as follows:

```
char chars [3];
int integer;
```

The character variable will still start at displacement 0. However, the integer variable will start at displacement 4 because integer data types are aligned on fullword boundaries. To make the alignment of the C structure the same as the alignment of the assembler structure, code it as follows:

```
typedef _Packed struct { char chars[3] ;
                         int  integer ;  } astruct ;
```

Note: The `struct` has no tag; this prevents unintentional unpacked declaration of this `struct`. With the MVS and OS/390 C/C++ compilers, you may use #pragma pack():

```
#pragma pack(packed)

struct astruct { char chars[3]; int integer; };

#pragma pack(reset)
```

Following are some reminders and coding techniques that you will find helpful when working with header files:

- The `#ifndef` directive can be used to prevent multiple declarations of the same identifier (in case the header file is included more than once). This allows headers to be included more than once without causing compile errors. The ISO/IEC C standard requires that all standard header files, except assert, can be `included` any number of times with the same effect as if they were only `included` the first time.
- The `#if` and `#ifdef` statements can be used to separate operating system-specific sections of code. For example, if you have an offline utility that runs differently on MVS systems than on TPF systems, but the interface as seen by the programmer is the same, these statements can be used to separate distinct code. The same file can then contain the source code for both.
- It is very important that for every `else`, `#if`, `#ifdef`, `#ifndef`, and `#elif` statement coded, there is a corresponding `#endif` in the same header file. Missing `#endif` statements can cause serious problems at compile time.
- The # and ## operators, for quotation and concatenation, are very useful when defining macros.
- Do not include header files inside any other declaration. In other words, include all header files at file scope. The ISO/IEC C standard specifically requires this for the headers that it describes.
- Define the user interfaces and symbols required by the interface in user API header files. The interfaces and symbols should not include any symbols that are used to implement the interface.
- Nesting should be done judiciously. Using several levels of nesting when declaring structures and unions has a serious drawback: the composite identifier names that must be used to refer to members of the structure or union at an inner nesting level can become quite lengthy and make programming and debugging unnecessarily challenging. When one level of nesting is not sufficient, you may want to declare separate unions or structures for each level, and use pointers to address them.
- The LSEARCH and SEARCH compile-time options control how the compiler locates filenames in <> and "" pairs under VM/CMS in the `#include` preprocessor directive. The SEARCH option is required for compiling TPF system programs, to insure that the correct header file library is present.
- You can view header files in a listing format by using the SHOWINC compile-time option.

TPF Application Environment

Accessing the ECB

There is an entry control block assigned to each input message or entry that defines all resources allocated to process that entry. The C data structure corresponding to the ECB is called `eb0eb`, and is defined in header file `c$eb0eb.h`. The macro, `ecbptr`, is used to obtain access to the ECB. Figure 19 shows 2 examples of using this macro. In the first example, pointer `ecb` is initialized to point to the current ECB. In the second example, `amsg` is a pointer data object that is assigned to point to the AM0SG record on data level 1. The `ecbptr` macro returns the base address of the ECB.
The `ecbptr` macro is resolved by a single instruction. Repeated calls to `ecbptr`, therefore, generate more efficient code than storing the result and using that result as a pointer to the `eb0eb` structure.

```
struct eb0eb   *ecb;
struct am0sg   *amsg;
   ⋮
ecb = ecbptr();            /* set ecb to point to ECB                */
amsg = ecbptr()->ce1cr1;   /* set aaa to point to 1st CBRW in ECB    */
```

Figure 19. Using `ecbptr` to Access the Contents of ce1cr1

Work Areas

The ECB contains areas for application use and reference, as well as several areas that only TPF can use. Areas available to the application program are known as work areas. There are 2 work areas, known as the *EBW* and *EBX* work areas, which are controlled by the application. These areas are used as transient work areas, as register save areas, for switch settings, or for other application purposes. C/C++ programmers, however, must use these areas judiciously, because they are subject to modification by any program segment that processes this entry. In the course of processing the entry, the contents of these areas are used for different purposes. Because automatic storage is available in C/C++ programming, it should be used for internal variables. For C/C++ programmers, the preferred purpose of the ECB work areas is to interface with assembler programs. There are several ways to pass parameters or values in assembler, including using registers and by using these ECB areas.

Programming Rule

Avoid using the ECB work areas for storing transient data. The preferred use of these areas is to pass arguments to or receive values from a called assembler segment. Use automatic storage for internal variables.

Data Levels

Information about file addresses is stored in 8-byte fields called *file address reference words* (FARWs). Information about certain main storage blocks is stored in 8-byte fields called *core block reference words* (CBRWs). There are 16 FARWs and 16 CBRWs per ECB, each of which is associated with a data level. The data levels are identified in hexadecimal notation as *D0* (data level 0) through *DF* (data level 15).

Think of a data event control block (DECB) as another ECB data level, but it does not reside in an ECB. An ECB data level and a DECB are very similar, but in a DECB the FARW has been expanded to 12 bytes to provide 8-byte file addressing. For more information, see “Data Event Control Blocks” on page 29.

Managing Files

The TPF system supports stream input/output (I/O) at an abstract level, with a hierarchical file system modeled on the UNIX and Portable Operating System Interface for Computer Environments1 standards (POSIX) TPF also supports record-level DASD I/O using TPF-specific FIND and FILE protocols. The TPF tape and general file interfaces are also record-level protocols.

TPF File System

The TPF file system automatically and transparently handles low-level problems such as record blocking, allocation, chaining, and locking protocols. The TPF file system also gives efficient direct access to data in a file. When used appropriately, the hierarchical file system can significantly reduce the cost and time required to design, implement, and maintain applications.

The TPF file system application programming interface (APIs) include all of the ANSI Standard C library (ISO/IEC 9899-1990, section 7) and most of the POSIX1, standards sections 5 and 6. The TPF file system is ***not*** POSIX compliant but for most of the API functions, the interfaces and semantics are identical to POSIX1 (see *TPF C/C++ Language Support User’s Guide* for details about deviations between the TPF file system APIs and the POSIX.1 standards).

The TPF file system APIs work at two levels:

- Buffered I/O
- System-level I/O.

The more abstract level is the buffered I/O functions, declared in <`stdio.h`>. The buffered I/O functions operate on a pointer to `FILE`. Examples include `fopen`, `fclose` and `fflush`; `fread` and `fwrite`; the formatted I/O functions such as `fprintf` and `fscanf`; the character I/O functions such as `fgetc`, `fgets`, `fputc`, `fputs`; and many others. These APIs are typically used in C language applications that need to create and access files. In general, they are easier to use and more efficient than the less abstract system-level functions that follow.

The system-level I/O functions, declared in <`unistd.h`>, <`fcntl.h`>, <`dir.h`>, and other headers, are appropriate in cases where an application needs to work on the directory structure itself or where more control over files than is provided by the buffered I/O functions is required. The system-level I/O functions operate on a file descriptor, a path name (which is a C string), or a pointer to `DIR`. Examples include `creat`, `open`, `close`, `read`, `write`, `fcntl`, `chdir`, `chown`, `chmod`, `mkdir`, `mknod`, and `stat`, and many others.

It is possible to switch levels if necessary. The `fileno` function returns the file descriptor underlying a pointer to `FILE`, allowing you to switch from buffered I/O to system-level I/O. Conversely, the `fdopen` function creates a pointer to `FILE` from a previously opened file descriptor, allowing you to switch from system-level I/O to buffered I/O.

Record-Level DASD I/O

The term *DASD* refers to direct access storage devices. C programmers can think of the term DASD as being synonymous with disk file, or disk, or file.

The basic record-level DASD I/O actions are implemented in TPF API functions `find_record` and `file_record`. The `waitc` function is used in TPF for error detection at the record I/O level. See *TPF C/C++ Language Support User’s Guide* for more information about these functions.

```
if(waitc())             /* if nonzero return code                          */
  {
  snapc(SNAP_EXIT,0x6404,"I/O ERROR",NULL,'U',SNAPC_NOREGS,SNAPC_ECB,NULL);
                                            /* force abnormal exit   */
  }
```

Figure 20. Using the `waitc` Function for Error Detection

There are two groups of find and file functions. The first group is *higher level* in the sense that invoking one function takes care of several different operations with one command. These functions are also simpler to use, because the programmer can specify certain fields (**id** and **rcc**) as parameters on the function call, rather than setting these CBRW fields with separate assignment statements.

`file_record`	File a record.
`file_record_ext`	File a record with extended options (this function includes TPF DECB support).
`find_record_ext`	Find a record with extended options (this function includes TPF DECB support).
`find_record`	Find a record.

Note: Applications that call the `find_record_ext` and `file_record_ext` functions and use 8-byte file addresses or DECBs in place of data levels, must be compiled with the C++ compiler. For more information about TPF DECB support, see “Data Event Control Blocks” on page 29.

Additional find and file functions that give the C programmer more control, and should, therefore, be used with more caution are:

`filec`	File a record
`filnc`	File a record with no release
`filuc`	File and unhold a record
`findc`	Find a record
`finhc`	Find and hold a record
`finwc`	Find a record and wait
`fiwhc`	Find and hold a record and wait
`unfrc`	Unhold a file record.

When using this second set of functions, programmers must have a greater understanding of the system file-record-level I/O routines and may need to call the `waitc` function to make sure the record is attached to the ECB.

See the detailed API function descriptions in *TPF C/C++ Language Support User’s Guide* for more information about the use of these functions.

The TPF system also uses another entity for file I/O, called a *general data set*. General data sets are organized on the basis of contiguous space on DASD, and are compatible with MVS. Two functions are used to manage this type of data set:

`gdsnc`	Open and close data set
`gdsrc`	Get record address.

Tape I/O

Real-time tapes are tapes that can be written to (and only written to) at any time by any operation in the system. There are two API functions available to write to real-time tapes:

`tourc`	Write record, release buffer block
`toutc`	Write record, retain buffer block.

General tapes are I/O tapes used for application programming. They allow the application program to write to consecutive files and read them in logical sequence. There are two groups of API functions for managing general tapes: basic general tape functions and high-level general tape functions.

A basic general tape function performs a single tape function and gives an ECB absolute control over a tape. (However, a tape can be shared between ECBs by assigning and reserving a tape in the appropriate sequence.) Using the basic general tape functions can also provide more efficient processing if it is needed. The basic general tape functions are:

`tasnc`	Assign tape to process
`tbspc`	Backspace tape
`tclsc`	Close a general tape
`tdspc`	Display tape status
`tdspc_q`	Display tape queue length
`topnc`	Open tape
`tprdc`	Read tape record
`trewc`	Rewind tape
`trsvc`	Reserve tape for other processes
`tsync`	Flush tape buffer
`twrtc`	Write tape record.

A high-level general tape function performs multiple tape functions from the set of basic general tape functions and allows all ECBs to share a tape.

At the beginning of processing a high-level general tape function, all the functions, except for `tape_open`, assign the tape. Therefore, when you use these functions, except for `tape_open`, you must reserve the tape or the tape must be in a reserved state from previous processing.

At the end of processing a high-level general tape function, all the functions, except for `tape_close`, reserve the tape. Therefore, when using these functions, except for `tape_close`, the tape will be left in a reserved state at the end of processing.

The high-level general tape functions are:

`tape_close`	Close a general tape
`tape_cntl`	Tape position control
`tape_open`	Open tape
`tape_read`	Read a record
`tape_write`	Write a record.

ECBs and Entries

In the TPF system each active entry has an ECB associated with it. Once created, the ECB is processed by a specific application, which can consist of many individual program segments.

Control

An ECB or entry has control when it has the attention of the CPU. Control is granted based on the CPU loop, the TPF system's scheduling system. (See *TPF Concepts and Structures* or *TPF Main Supervisor Reference* for more detail about the CPU loop and task dispatching.)

Control is lost under the following circumstances:

- When some system services are accessed (for example, find, wait, and hold). The nature of the request determines when control is returned to the entry.
- Exit processing. This can occur voluntarily, by calling either the `exit` or `abort` functions, or involuntarily, as the result of the entry causing a system error.

The TPF system will also force an ECB to exit if it does not relinquish control to the operating system in 500 milliseconds (that is, it appears to be looping). This timer is reset when an ECB waits for pending I/O with the `waitc` function and when it suspends processing through the `dlayc` and `defrc` functions.

When TPF Enter/Back services are invoked, the same ECB remains in control and continues processing with another program segment.

You can create an entry (ECB) by calling one of the following TPF API functions:

`credc`	Create a deferred entry
`creec`	Create a new ECB with an attached block
`cremc`	Create an immediate entry
`cretc`	Create a time-initiated entry
`cretc_level`	Create a time-initiated entry with an attached core block
`crexc`	Create a low priority entry.
`tpf_cresc`	Create a synchronous entry
`system`	Execute a command

The application program is responsible for passing any required data to the newly created ECB when it calls the TPF API library function. In particular, note that the input message and terminal address are not automatically copied to the new ECB. In the TPF system environment, there is not a strong parent-child relationship between entries; the new ECB cannot communicate anything back to the ECB that created it unless the `tpf_cresc` or `system` function is used, whereby the child ECB may pass or return a value back to the parent ECB.

Use caution when using the ECB creation functions. There is a limit to the number of entries that can be active in the system and approaching this limit too closely can degrade performance.

Exit Processing

When an entry is completed, control is returned to the operating system. If the entry was created by the `system` function calling a DLM with a `main` function, or if it was created by the CRESC macro or the `tpf_cresc` function, the system adds the parent process to the CPU ready list so that it can resume processing.

The entry's exit status is returned to the parent process as the `system` function's return code. Returning from the initial call to a `main` function is equivalent to calling the `exit` function with the return value as its status parameter. For example:

```
int main(void) { return 15; }
```

is equivalent to:

```
#include <stdlib.h>
int main(void) { exit(15); }
```

If the initial program that the entry ran does not contain a `main` function, the entry must explicitly call a process-terminating function, such as `exit`, `abort`, `serrc_op`, `snapc`, and others to return control to the operating system. The `abort` function forces the current ECB to exit under abnormal circumstances; no dumps are issued for invalid ECB states such as having a hold on a file address, or having a general tape opened or assigned. The `serrc_op` function causes a system error dump to be generated. Following the dump the ECB will be exited if the defined term **SERRC_EXIT** was coded as one of the parameters. The `snapc` function causes a system error dump to be generated. Following the dump the ECB will be exited if the defined term **SNAPC_EXIT** was coded as one of the parameters.

TPF Terminal Communications

Terminal Input

The user input terminal can only send messages to the system, which is then responsible for scheduling a process associated with the message. TPF application programs are activated by these input messages. Although they can send output messages in response, they do not remain active and wait for additional input messages except for TPF/APPC where the TP remains active in the same ECB throughout the life of a conversation. For non-TPF/APPC messages each new input message that arrives will cause the TPF system to create a new, independent ECB.

Therefore, non-TPF/APPC application programs must obtain all information required to process the transaction from the single input message along with data contained in the TPF file system. TPF application programs cannot operate in an interactive mode because they are not able to “listen” for additional input messages.

Programming Rule

Use the parser (IPRSE) or `sscanf` to obtain portions of the input message. Do not attempt to interact with the user input terminal. Design applications in such a manner that all information required to process the transaction is available in either the input message itself or in existing data records available to the process.

When a user enters a message, it is processed by the appropriate network facilities and passed to the TPF system. The message is then added to the input list (processed by the CPU loop), and becomes associated with data level 0 (D0). The standard C function `sscanf` has been adapted to the TPF system to access these user input messages from D0. The only difference is that these functions do not solicit terminal input, but will access the message on D0. If the user attempts to get input message lines that do not exist, a null will be returned. Likewise, if the message associated with D0 has been released or is not an input message, a null will be returned.

Output Messages

There are several different options available to C and C++ programmers who want to send an output message to the user terminal:

I/O stream	An I/O stream, including `stdout` or `stderr` can be opened on a special file node such as `/dev/tpf.omsg/`, which is associated with a device driver that writes an output message. Any of the formatted I/O functions (`fprintf`, `printf`) or text I/O functions (`fputs`, `putc`, `putchar`, `puts`) can then be used to write the output message to the I/O stream.
`routc`	Requires a routing control parameter list (RCPL) to be built and passed to it as an argument. An RCPL is associated with each input and output message, and identifies the origin, destination, and characteristics of the message. For this reason, it is recommended that an application program preserve the RCPL received at activation time and change it to an output RCPL (by swapping the origin and destination addresses). The `routc` function can be used with either single line or full-screen formatted output (SNA) terminals. Also, TPF Advanced Program-to-Program Communication support allows TPF application programs to communicate with LU 6.2 applications on remote platforms. (For more information about TPF Advanced Program-to-Program Communications support, see *TPF ACF/SNA Data Communications Reference*.) The TPF system also supports direction of output using a device driver. A device driver should be designed by the user to output streams to the desired terminal.

Using TPF Globals

There must be a tag name for each global field and record defined in the `c$globz.h` header file. This file is included by `tpfglbl.h`, which is required whenever you access global fields or records. Whenever changes are made to `c$globz.h`, you must recompile any programs that access the global tags.

There are 2 functions that you can use to access TPF globals:

`glob`	Returns the address if a field is specified and returns the address of global directory entry if record is specified.
`global`	Allows you to modify a global field or record.

The following functions are available in ISO-C only.

`glob_keypoint`	Keypoints a global field or record.
`glob_lock`	Locks and accesses a global field or record in preparation for a synchronous update.

`glob_modify`	Modifies a global field or record.
`glob_sync`	Synchronizes a global field or record across a complex.
`glob_unlock`	Unlocks a global field or record.
`glob_update`	Updates a global field or record.

The `glob` function provides read-only access, whereas the `global` function allows you to update, modify, keypoint, lock (reserve for exclusive use), unlock, copy, or synchronize global fields. These command options must be run in the correct order; for example, you cannot issue an UNLOCK before you have issued a LOCK. For more detail about these options, see *TPF C/C++ Language Support User's Guide*.

The global tags acted on by C functions must be defined in the `c$globz.h` header file or unpredictable results can occur.

Naturally, locks should not be held longer than necessary. Programs using the `glob_lock` function should be prepared to call `glob_update`, `glob_sync` or `glob_unlock` as soon as possible to prevent severe system degradation caused by other ECBs waiting for the lock.

If the global field or record can be synchronized, `glob_lock` must be called before calling `glob_modify` or `glob_update`.

Programs using these functions should not have any pending I/O operations outstanding because they can perform the equivalent of a `waitc` function.

You can find more detailed information about TPF globals as follows:

Type of Information	Reference
High-level description	"Creating Globals for C" on page 73
Installation details	"Customizing C/C++ Language Support" on page 301
Basic TPF global concepts, terminology, overview	*TPF System Installation Support Reference*
Global program logic	*TPF System Installation Support Reference*.

Calling Other Functions and Programs

The following TARGET(TPF) calls are not supported:

- Library functions cannot call entry points, including assembler programs.
- Assembler language programs cannot call library functions or static functions.
- TARGET(TPF) cannot issue `system` calls

The compilers that support the TARGET(TPF) compiler option also provide the appropriate linkage for other function calls. Ensure that you do the following:

1. You must include the `tpflink.h` header file by using `#include`. This file contains the `#pragma` directive for the library.
2. Parameter passing.

Function Linkage:

In ISO-C the linkage type is automatically defined by the linkage editor or online service.

#pragma Compiler Directive for TARGET(TPF)

A `#pragma` directive is an implementation-defined instruction to the compiler used for assembly language programs or library functions in TARGET(TPF). When a TPF program is compiled, the `#pragma` directive is used to communicate critical information to the compiler concerning:

- The type of function being compiled
- The type of any function called by the function being compiled.

When a C function is compiled for use in the TPF system environment, the compiler needs to know which TPF program segment will contain the compiled code. For entry points, the function name is initially assumed to match the TPF segment name. However, from the C programmer's point of view this is not always desirable, and there is the additional problem of the library functions, of which none of the names match TPF segment names.

The compiler is notified of the segment name to which a function name maps by the `#pragma map` directive. Figure 21 shows the format of the `#pragma map` statement.

```
        #pragma map(internal_name, "external_name")
```

#pragma
is the compiler directive used to pass information to the compiler

map
identifies the type of data being passed

internal_name
is the name by which the C function is known

external_name
is the name used when generating linkage to the function.

Figure 21. TARGET(TPF) C #pragma map Statement Format

For example, the statement

```
#pragma map(accept_trans, "QZZ0")
```

maps the TPF segment name QZZ0 to C function `accept_trans`.

When the compiler finds a call to another C function, the compiler needs to know what type of function call expansion to generate. When compiling the called function, the compiler needs to know what type of return linkage to generate.

The `#pragma linkage` with type TPF statement is not supported in ISO-C and should be removed from any TARGET(TPF) programs that are being migrated to ISO-C. In TARGET(TPF), linkage information is passed to the compiler by the `#pragma linkage` directive. Figure 22 on page 158 shows the format of the `#pragma linkage` statement for TARGET(TPF). `#pragma linkage` of type TPF is not valid for ISO-C.

```
#pragma linkage(internal_name,TPF,tpftype)
```

#pragma
is the compiler directive used to pass information to the compiler.

linkage
identifies the type of data being passed.

internal_name
is the name by which the C function is known.

TPF
is the string that identifies the TPF programming environment. This is valid for TARGET(TPF) only.

tpftype
describes the type of object being linked to:

1. C indicates a C entry point. This is the default type used if no linkage statement is found.
2. N indicates that the function is an assembler program.
3. An integer in the 0–999 range indicates that the function is a library function.

Note: This value is not range-checked at compile time.

Figure 22. C #pragma Linkage Statement Format for TARGET(TPF)

For example, the statement

```
#pragma linkage(malloc, TPF, 42)
```

identifies `malloc` as a library function and assigns it an index number of 42 in the quick enter directory.

If no `#pragma linkage` directive is found for a given function, the compiler will assume a type-C linkage.

All of the `#pragma` directives required for the C/370 run-time library programs are collected into a single header file, `tpflink.h`. The `tpflink.h` header is included by `tpfeq.h`. Because `tpfeq.h` is required to be the first header in every TARGET(TPF) C program, all linkage information for the C/370 library functions required by the compiler is guaranteed to be available to application programs.

Parameter Passing (from C to C)

There are no special considerations for parameter passing between C functions or programs in TPF systems, except for the passing of structures.

Note: The following information is a TARGET(TPF) restriction only. Although structures are passed by value (as are all parameters), this can take up a lot of stack storage, and stack storage is a scarce commodity. One way to preserve stack storage is to pass addresses of structures (using the & operator), rather than passing the structures themselves. The parameter should be declared as *const* in the called function (to prevent unintentional modification of the structure passed by reference). This practice is strongly recommended.

Calling a C Program (from Assembler)

There can be times when you want to call a segment written in C from a segment written in assembler. This is done using an Enter function. When doing so, it is important to know whether the C segment being called is TARGET(TPF) or ISO-C because the register conventions are different in each. See *TPF Program Development Support Reference* for more information about register conventions.

Note: The preceding information does not apply to C++ functions except for those that use `extern "C"` linkage.

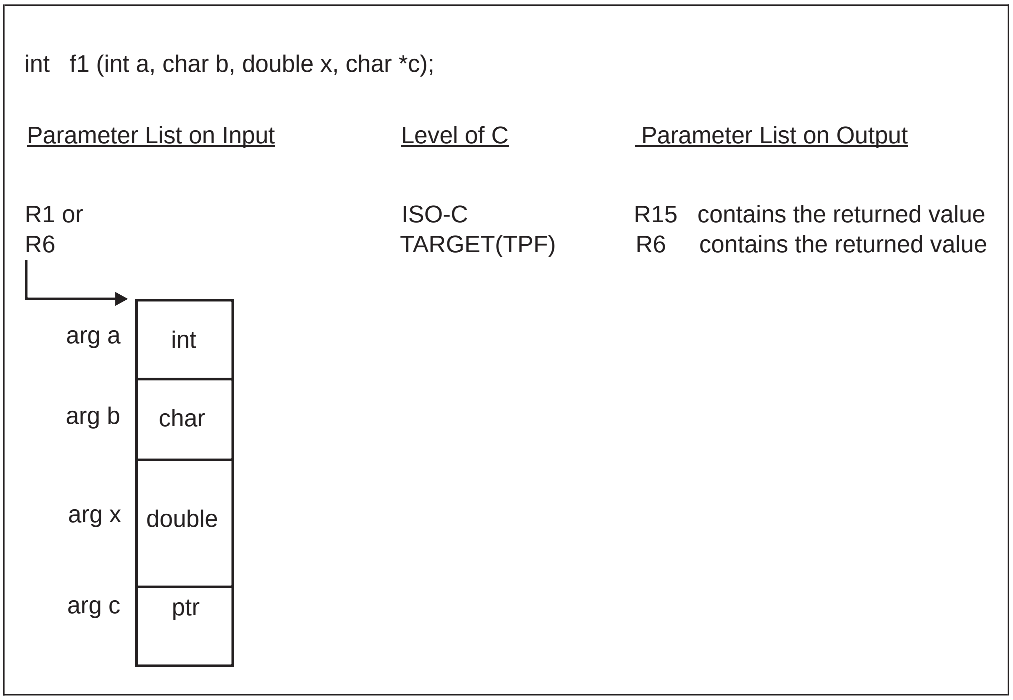
Assembler Calling C: The C Parameter List

Because C entry points are normally called using the TPF Enter mechanism, it is possible to start a C entry point function from an assembler program using one of the Enter macros. The assembler program must construct a C parameter list in the format expected by the C compiler-generated code. This section explains the format of the C parameter list for ISO-C and TARGET(TPF).

Note: The format of the C parameter list is an IBM restricted interface, and could change in future releases. Do not use this interface unless absolutely necessary.

The compiler-generated code allocates storage for the parameter list in the calling function's stack frame. All parameters are passed by value. This applies to structures as well; the entire structure is copied to parameter list storage. (Arrays are treated as pointers; the *address* of the array is passed by value.)

The compiled code assumes that R1 contains the address of the parameter list for ISO-C and R6 contains the address of the parameter list for TARGET(TPF). The format of the parameter list takes one of two forms, depending on the type of value returned by the function. If the function returns type integer, character, or pointer, the parameter list has the structure shown in Figure 23.



```
int   f1 (int a, char b, double x, char *c);
```

Parameter List on Input	Level of C	Parameter List on Output
R1 or	ISO-C	R15 contains the returned value
R6	TARGET(TPF)	R6 contains the returned value

Figure 23. C Parameter List (Part I)

Each parameter occupies one fullword of storage, except for float and double, which occupy two consecutive fullwords. Unsigned parameter values are right justified and padded on the left with binary zeros. Signed parameters are right justified and have sign extension on the left.

If the function returns type `float`, `double`, `struct`, or `union`, the parameter list has the structure shown in Figure 24.

The first fullword of the parameter list contains the address of an area of sufficient size to contain the returned float, double, structure, or union. The called function simply stores its result in the area indicated.

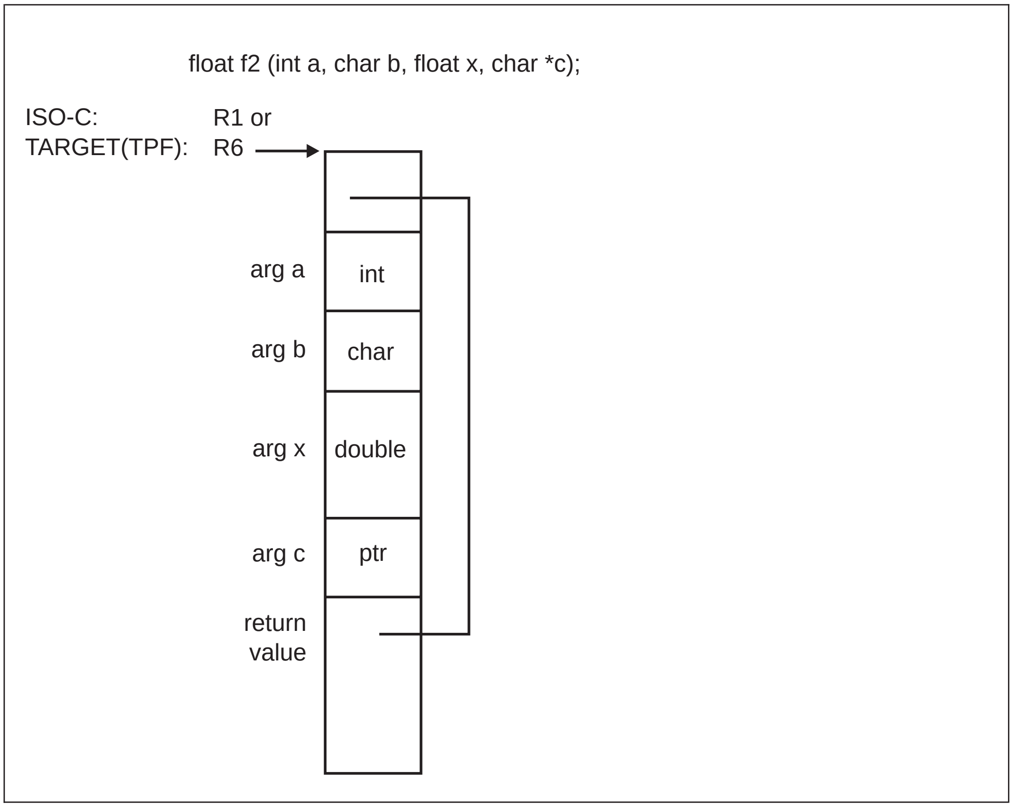


Figure 24. C Parameter List (Part II)

Calling an Assembler Program (from C)

This section describes the interface requirements for assembler and C language.

Entry points implemented in assembler language can be called from ISO-C without any special calling protocol. When the assembler segment is called, the run-time parameter linkage is automatically handled by the control program.

```
struct  TPF_regs
  {
  long int    r0;
  long int    r1;
  long int    r2;
  long int    r3;
  long int    r4;
  long int    r5;
  long int    r6;
  long int    r7;
  };
```

Figure 25. `TPF_regs` Structure. Used to pass parameters to assembler programs in registers.

In TARGET(TPF), entry points (TPF segments) written in assembler are typically called *N-type linkage* segments, which comes from their required `#pragma linkage` statement:

```
#pragma linkage(SEG1, TPF, N)
```

The N in this statement means Non-C. N-type linkage implies that the segment to be called has no knowledge of the existence or format of a C compiler parameter list and may, therefore, be expecting some or all of its parameters in registers.

In either ISO-C or TARGET(TPF), to call an assembler program in C language, you must code a valid C prototype statement for the assembler program, passing the `TPF_regs` structure as follows:

```
void seg1(struct TPF_regs *);
```

Including this prototype statement allows the compiler to do type checking during compilation, and instructs the compiler not to generate or expect any return value. These prototype statements are declared as returning type `void`, although in ISO-C a NULL pointer can be passed instead of the register structure. Even though C prototype for assembler programs specify void return, assembler program's R0–R7 are returned in TPF_regs. Because most assembler programs are unaware of the mechanisms of returning values to a C calling function, declaring another return type can cause unpredictable results.

The `TPF_regs` structure is predefined in `tpfregs.h`, as shown in Figure 25. `TPF_regs` can be used in either ISO-C or TARGET(TPF).

Whenever an assembly language program is called, the only argument ***must*** be a pointer to type `TPF_regs`.

Programming Rule

When calling an assembly language program declare prototype statements to take 1, and only 1, argument: a pointer to a structure of type `TPF_regs`. For ISO-C a null pointer can be coded for `TPF_regs`.

TARGET(TPF) Restriction

If the N-type linkage function does not take any arguments in registers, allocate the space for a `TPF_regs` structure and pass a pointer to it anyway. This is necessary because the linkage code uses the second part of the `TPF_regs` structure to save the calling C program's registers (R0–R5) to protect its environment from all possible calling sequences. Also, before returning control to the calling C program, the contents of registers 0–7 are stored in the same location (the `TPF_regs` structure) so that the C program has access to any return values from the assembler program.

Compiling and Running C/C++ Programs

This section contains a brief description of compiling and running C or C++ programs. See the user's guide for the IBM C or C++ compiler on the System/390 platform used by your installation for more detailed information about compiling C and C++ application programs.

The following briefly describes the RENT and LONGNAME compiler options.

RENT

Causes the compiler to generate reentrant code. Reentrancy of a single variable can be controlled by coding

```
#pragma variable( vbl, RENT )
```

RENT must be used if the program uses writable static. The RENT directive is not required if the program being compiled does not use writable static. If RENT is not used and writable static is encountered, a system error occurs. If RENT is used and there is no writable static, there is a performance loss associated with secondary linkage.

The RENT/NORENT option and `#pragma` variables are not supported for TARGET(TPF).

This parameter is assumed if the TARGET(TPF) option is used.

Note: There is no RENT compiler option in the family of IBM C++ compilers on the System/390 platform. Source code written in C++ and compiled with a C++ compiler will automatically be compiled as RENT.

See *TPF Programming Standards* for more information about the TPF RENT standard. See "Sample Code Written to the RENT Standard" on page 163 for code samples.

LONGNAME/NOLONGNAME

Enables or suppresses support for long and mixed case external variable names. This directive is not supported for TARGET(TPF).

Compiling with the LONGNAME option allows you to use external names that are unique in the first 255 characters, with case respected. For code compiled with the NOLONGNAME compiler option, all identifiers that have external linkage must have names that are unique in the first 8 characters, ignoring case.

Segments that will be linked into a single load module must be compiled either all with the LONGNAME option or all with the NOLONGNAME option.

See *TPF Programming Standards* for more information about the LONGNAME standard.

DLL Compiler Option

DLL applications written in C are compiled with the DLL compiler option. For this case, function pointers are pointers to corresponding function descriptors. Therefore, a function pointer that is passed from code compiled without the DLL option to a function in code compiled with the DLL option will not work because the DLL code always expects a function descriptor pointer. Code compiled with the DLL option can still pass a function pointer to code compiled without the DLL option. The special code at the beginning of the function descriptor handles these conditions. See the user's guide for the IBM C/C++ compiler on the System/390 platform used by your installation for more information about the DLL compiler option and function descriptors.

There is no DLL compiler option for source code written in C++. C++ source code is automatically compiled as a DLL application.

Sample Code Written to the RENT Standard

The following 3 sample code examples correlate to the RENT standard described in *TPF Programming Standards*.

C Header File with Declarations of External Linkage Objects

Sample 1

```
/**********************************************************************/
/* c$xmp1.h:  header file showing declarations of external linkage    */
/*            objects.                                                */
/**********************************************************************/

/***************************************/
/* Declarations of writable externals. */
/***************************************/

extern int i;

#define NUM_ELEMENTS 3
extern int ia[NUM_ELEMENTS];

extern const char *ccp;         /* non-const pointer to const char */

/**********************************************************************/
/* Declarations of NORENT read-only externals.  There are various     */
/* reasons that you may need to specify NORENT externals, including:  */
/*                                                                    */
/* 1. Defining externals in a runtime library which cannot contain    */
/*    RENT static for performance reasons.                            */
/*                                                                    */
/* 2. Referring to an external which is defined in assembler.         */
/*                                                                    */
/* Note that #pragma variable does not work on static variables with  */
/* internal linkage (i.e. declared with the "static" keyword) or      */
/* string literals (such as "Hello, world!\n"); for these you need    */
/* the NORENT compile time option or #pragma strings(readonly).       */
/*                                                                    */
/* Note also that all NORENT variables must be read only and should   */
/* therefore be declared using the "const" keyword.                   */
/**********************************************************************/
```

```
#pragma variable(const_i,NORENT)
extern const int const_i;

#pragma variable(const_ia,NORENT)
extern const int const_ia[NUM_ELEMENTS];

#pragma variable(cp_const,NORENT)
extern char * const cp_const;   /* const pointer to non-const char */
```

C Source File with Definitions of External Linkage Objects

Sample 2

```
/*********************************************************************/
/* xmp1d.c:  C source file showing definitions of external linkage    */
/*           objects.                                                 */
/*                                                                    */
/* Note:  This module is not naturally reentrant because of the       */
/*        modifiable static duration objects that it defines, so it   */
/*        must be compiled with the RENT option.  It also defines a   */
/*        writable static object with internal linkage (see variable  */
/*        sca below).                                                 */
/*********************************************************************/

/*********************************************************************/
/* The header file (above) containing the external declarations is    */
/* included to guarantee that the definitions and the RENTness are    */
/* consistent with the declarations which are #included in source     */
/* files that reference these externals.                              */
/*********************************************************************/
#include <c$xmp1.h>             /* include declarations of externals */

/**********************************/
/* Define the writable externals. */
/**********************************/
int i = 17;
int ia[NUM_ELEMENTS] = { 0, 1, 2 };
const char *ccp = "ccp can be changed to point to another string.";

/********************************/
/* Define the NORENT externals. */
/********************************/
const int const_i = 42;
const int const_ia[NUM_ELEMENTS] = { 0xC0, 0xC1, 0xC2 };
static char sca[] = "cp_const will always point to this array, but the
                    "contents of the array can change.";
char * const cp_const = sca;
```

C Source File Showing the Use of External Linkage Objects

Sample 3

```
/*********************************************************************/
/* xmp1u.c:  C source file showing use of external linkage objects.   */
/*                                                                    */
/* Note:  This module does not define any static duration objects,    */
/*        but it does REFER to the static duration objects which are  */
/*        compiled with the RENT option; therefore, it must also be   */
/*        compiled with the RENT option.                              */
/*********************************************************************/

#include <string.h>
#include <c$xmp1.h>             /* include declarations of externals */

int foo(void)
{
    /**************************************************************/
    /* Only the RENT/writable (non-const) variables can be changed. */
    /**************************************************************/
    for (i = 0; i < NUM_ELEMENTS; ++i)
    {
        ia[i] = const_ia[i];
    }
    strcpy(cp_const, ccp);      /* change the non-const chars */
    ccp = cp_const;             /* change the non-const pointer */
    return const_i;
}
```

Understanding TPF Internet Server Support

This chapter describes:

- The functions of the Portable Operating System Interface for Computer Environments (POSIX) process model that are implemented by the TPF system
- The Internet daemon in the TPF system
- How Internet server applications run on the TPF system
- How to start a TPF application from the Internet
- Administering the TPF system as a Web site.

The POSIX Process Model As Implemented by the TPF System

TPF Internet server support allows POSIX-compliant servers (Internet server applications) to be written for or ported to the TPF system, usually with only minimal modifications to account for the differences in the TPF implementation. Because of the existing unique architecture of the TPF system, TPF Internet server support consists of a subset of POSIX-compliant application programming interface (API) functions and TPF-unique API functions.

Note: The TPF system is not POSIX-compliant; only a subset of POSIX-compliant API functions are implemented in the TPF system.

The following discussions assume a basic knowledge of client, server, and UNIX concepts. Suggested references that describe these concepts are:

- *UNIX Network Programming*
- *UNIX Network Programming: Networking APIs: Sockets and XTI*.

A Process

A *process* is an address space and the single thread of control that executes within that address space and its required system resources. There is a process block associated with each entry control block (ECB) where process-related information is maintained by the TPF system.

In the TPF system, all active ECBs are part of a process. The `system`, `tpf_cresc`, and `tpf_fork` functions create a child process.

See the *TPF C/C++ Language Support User's Guide* for more information about the `system`, `tpf_cresc`, and `tpf_fork` functions.

Process ID

A *process ID* is a unique, positive number that represents a process. Because the process ID is a unique identifier, it can be used to direct signals between processes. See "Signals" on page 169 for more information about signals. The process ID is also used in the `tmpnam` function; this function returns a temporary file name based on this unique identifier.

Every process also has a parent process ID associated with it. If there is no parent process, the parent process ID is set to 1. For a child process created by the `tpf_fork` function, the parent process ID is set to the process ID of the parent process.

A process ID is a unique two-part number, 32 bits in length:

- The first part is an index that corresponds directly to the address of an entry control block (ECB); the details of the correspondence are not important, but knowing that the correspondence exists is important.
- The second part is a counter that is incremented each time an ECB is reused. The counter reduces the risk of using a leftover process ID of an ECB that has been reused.

The `getpid` and `getppid` functions are provided so that an ECB can obtain its process ID and the process ID of its parent process, respectively. See the *TPF C/C++ Language Support User's Guide* for more information about the `getpid` and `getppid` functions.

Process Group

A *process group* is a set of related processes and each process has a process group ID associated with it. Most processes, when created, are assigned to their own process groups. However, a child process created by the `tpf_fork` function is part of the process group of its parent process, so the child process inherits the process group ID of its parent process.

Process Inheritance

A child process created by the `system` and `tpf_cresc` functions inherits the environment list from the parent process. A child process created by the `tpf_fork` function inherits the following properties from the parent process:

- Environment list
- Real user ID (UID) and real group ID (GID)
- Process group ID
- Current working directory
- File mode creation mask
- Open file descriptors that do not have the `FD_CLOEXEC` file descriptor flag set to 1.

The effective user ID and effective group ID of a child process created by a `tpf_fork` function are initialized to the effective user ID and effective group ID of the parent process, with the following exceptions:

- If TPF_FORK_FILE is specified and if the file specified by the **program** parameter has the set-user-id (S_ISUID) flag set, the effective user ID of the child process is set to the user ID of the *owner* of the file.
- If TPF_FORK_FILE is specified and if the file specified by the **program** parameter has the set-group-id (S_ISGID) flag set, the effective group ID of the child process is set to the group ID of the *owner* of the file.

The saved set-user-ID and saved set-group-ID of a child process created by the `tpf_fork` function are initialized to the same values as the effective user ID and effective group ID respectively. This enables an application to toggle its effective user ID between the real user ID and the owner ID of the executable file, if applicable.

POSIX-Compliant APIs for Process Control

The following functions have interfaces identical to the POSIX standards. However, because of the unique TPF architecture, there are differences in the way the functions behave; refer to the individual functions in the *TPF C/C++ Language Support User's Guide* for specific information about the differences.

If Internet server application code written for the UNIX system is ported to the TPF system, these function calls do not necessarily need to be changed, but it is likely that the Internet server application needs some modification to account for the differences.

- `alarm`
- `getpid`
- `getppid`
- `kill`
- `pause`
- `raise`
- `sigaction`
- `signal`
- `sigpending`
- `sigprocmask`
- `sigsuspend`
- `sleep`
- `wait`
- `waitpid`.

TPF-Unique APIs for Process Control

Because of the unique TPF architecture, there are some functions that cannot be implemented with interfaces identical and behavior similar to the POSIX standards. The following are the TPF-unique functions for process control:

- `tpf_fork`, which is based on the POSIX `fork` and `exec` functions
- `tpf_process_signals`, which is based on POSIX signal support.

  Because the TPF system does not allow a process to be interrupted asynchronously, the `tpf_process_signals` function is a way for a process to explicitly tell the TPF system to check for and handle outstanding signals.

Signals

A *signal* is a simple method of communication between two processes and is used for processes to communicate with each other about important events.

In the POSIX process model, signals are sent to processes by the system for a number of reasons, synchronously and asynchronously. The SIGCHLD signal is the only signal sent by the TPF system. Additionally, a subset of POSIX signals are implemented in the TPF system and are listed in the description of the `signal` function in the *TPF C/C++ Language Support User's Guide*.

Signal Handlers

Any process can send a signal to any other process if the sending process knows the process ID of the intended receiving process and if one of the following is true:

- The real user ID or effective user ID of the sending process matches the real user ID of the receiving process
- The real user ID or effective user ID of the sending process matches the saved set-user-ID of the receiving process
- The sending process has superuser privileges.

Sending a signal does not imply that any action will be taken by the receiving process. In the TPF system, incoming signals are only handled by a process when the process explicitly requests that signals be handled. The receiving process can choose to process signals, selectively ignore signals, or not process signals.

The following APIs are used to request the handling of pending signals:

- `sleep`
- `tpf_process_signals`
- `wait`
- `waitpid`.

For example, if signals are used by an application to allow a monitoring process to shut down other processes in the application, the monitoring process could do so only if all of the programs in the application periodically issue a `sleep`, `tpf_process_signals`, `wait`, or `waitpid` function.

SIGCHLD Signal

The SIGCHLD signal is the only signal that the TPF system sends to a process. When a child process created by the `tpf_fork` function ends, the TPF system:

- Sends a SIGCHLD signal to the parent process to indicate that the child process has ended
- Saves the exit status of the child process so that the parent process can identify which child process (by process ID) ended and its exit status.

Exit Status

Exit status is saved only if the parent process still exists and it handles signals; that is, the SIGCHLD signal disposition is not set to ignore signals (the SIGCHLD signal handler is not set to SIG_IGN).

An application (especially a long-running application) that creates many child processes using the `tpf_fork` function must periodically check for saved exit status so that the system resources used to save exit status can be freed. If signals are not processed on a periodic basis, there is a potential to deplete system resources. Alternatively, if an application creates child processes using the `tpf_fork` function, but is not interested in knowing when its child processes end, the application can notify the TPF system to ignore SIGCHLD signals, and as a result, the exit status for the child processes is not saved and the system resources are not tied up.

File Access in the TPF File System

Access to a file in the file system is POSIX-compliant and is controlled by the effective user ID, effective group ID, and access permissions.

Process Attributes

There are process attributes that are used to determine whether a process can access a file:

- The *effective user ID* is a user ID associated with the last `setuid` or `seteuid` function.
- The *effective group ID* is a group ID associated with the last `setgid` or `setegid` function.

Access Permissions

Associated with a file in the file system are *access permissions* that determine if a process can access a file. The access permissions available for a file are read, write, and execute (or any combination), and can be set for a file owner (or user), group, and users other than the owner or group. Users other than the owner or group are referred to as just *other*.

If the bit corresponding to the action that a process wants to take is on, file access is granted. Use the `chmod` function in a program to change the access permissions of a file. Use the ZFILE chmod command to manually change the access permissions of a file. Enter the ZFILE ls command with the -l parameter specified to display the access permissions of a file.

Table 17 shows a summary of the different access permissions and the settings used by the `chmod` function and the ZFILE chmod and ZFILE ls commands.

Note: The access permission values used by the ZFILE chmod command are in octal notation.

Table 17. Access Permissions

Accessed By	**Access Type**	**`chmod` Function Settings**	**ZFILE chmod Command Settings**	**ZFILE ls Command Settings**
User	Read	S_IRUSR	0400	`r--------`
	Write	S_IWUSR	0200	`-w-------`
	Execute	S_IXUSR	0100	`--x------`
Group	Read	S_IRGRP	0040	`---r-----`
	Write	S_IWGRP	0020	`----w----`
	Execute	S_IXGRP	0010	`-----x---`
Other	Read	S_IROTH	0004	`------r--`
	Write	S_IWOTH	0002	`-------w-`
	Execute	S_IXOTH	0001	`--------x`

See *TPF C/C++ Language Support User’s Guide* for more information about the `chmod` function. See *TPF Operations* for more information about the ZFILE chmod and ZFILE ls commands.

Rules to Determine File Accessibility

The following rules determine if a process can access a file:

- The effective user ID of the process is compared to the owner of the file.

  If they match, the user access permissions are checked.

  – If the user access permission associated with the action that the process wants to take is on, the process is allowed to take the action; that is, file access is granted.

  – If the user access permission associated with the action that the process wants to take is off, access is denied.

- If the effective user ID of the process does not match the owner of the file, the effective group ID of the process is compared to the group of the file.

  If they match, the group access permissions are checked.

  – If the group access permission associated with the action that the process wants to take is on, the process is allowed to take the action; that is, file access is granted.

  - If the group access permission associated with the action that the process wants to take is off, access is denied.
- If the effective user ID of the process does not match the owner of the file and the effective group ID of the process does not match the group of the file, the other access permission is checked.
  - If the other access permission associated with the action that the process wants to take is on, the process is allowed to take the action; that is, file access is granted.
  - If the other access permission associated with the action that the process wants to take is off, access is denied.

Use the ZFILE chmod and ZFILE chown commands to maintain file accessibility by changing the access permissions and the owner or group of a file in the file system. See *TPF Operations* for more information about ZFILE chmod and ZFILE chown commands.

Internet Daemon

The Internet daemon consists of two major components:

- The Internet daemon *monitor*, which is responsible for starting and stopping the Internet daemon listeners for Internet server applications and for error recovery when an Internet daemon listener fails.
- An Internet daemon *listener*, which monitors the Internet server applications and, with some process models, creates and monitors a socket for the Internet server application.

Process Models

The following process models define the interface to the Internet daemon:

- The WAIT process model offers synchronous control by using the `tpf_fork` function to create a child process. When the entry control block (ECB) associated with the child process ends, the TPF system sends a SIGCHLD signal to the Internet daemon (parent process). After the Internet daemon issues the `tpf_fork` function, the Internet daemon waits for the child process to end before activating another child process. In other words, only one child process can be active at a time.

  For TCP servers, the child process is created when a remote client connects and the Internet daemon accepts the socket connection. For UDP servers, the child process is created when a message is received on the UDP socket. Information about the socket is passed to the child process, enabling the child process to communicate with the remote client over the socket.
- The NOWAIT process model offers synchronous control by using the `tpf_fork` function to create a child process. When the ECB associated with the child process ends, the TPF system sends a SIGCHLD signal to the Internet daemon (parent process). Multiple child processes can be concurrently active up to a user-defined limit. You can define this limit by specifying the MAXPROC parameter on the ZINET ADD or ZINET ALTER command. See *TPF Operations* for more information about the ZINET ADD and ZINET ALTER commands.

  For TCP servers, the child process is created when a remote client connects and the Internet daemon accepts the socket connection. For UDP servers, the child process is created when a message is received on the UDP socket. Information about the socket is passed to the child process, enabling the child process to communicate with the remote client over the socket.

For UDP servers, after the Internet daemon creates a child process, the next child process is not activated until the previous child process ends or the previous child process sends a SIGUSR1 signal to the Internet daemon indicating that the child process is no longer using the socket. If child processes do not issue any SIGUSR1 signals, the NOWAIT process model has the same characteristics as the WAIT process model for UDP servers.

- The AOR process model offers asynchronous control for TCP servers by using the `activate_on_receipt` or `activate_on_receipt_with_length` function. When a remote client connects, the Internet daemon issues an `activate_on_receipt` or `activate_on_receipt_with_length` function to pass control of the new socket to your TCP server application when the first message is received from the remote client. After the Internet daemon issues the `activate_on_receipt` or `activate_on_receipt_with_length` function, the Internet daemon continues processing.
- The NOLISTEN and RPC process models offer no control because a `swisc_create` function is used to create an ECB for the specified Internet server application. The Internet daemon is used only to activate the server for these models. The Internet daemon does not create or monitor any sockets, nor does it monitor the server application.
- The DAEMON process model is similar to the NOLISTEN process model, but uses the `tpf_fork` function. The DAEMON process model does not create or monitor any sockets, but the Internet daemon ***does*** monitor the server application.

All Internet server applications handled by the Internet daemon follow a defined process model; these process models are described in more detail in “Internet Server Application Considerations” on page 176 and “Considerations for Using the Internet Daemon to Start a TPF Program” on page 188.

Internet Daemon Configuration File (IDCF)

The data for all Internet server applications processed by the Internet daemon is maintained in the Internet daemon configuration file (IDCF). See *TPF Transmission Control Protocol/Internet Protocol* for more information about the IDCF.

Hypertext Transfer Protocol (HTTP) Server

Although an HTTP server is not provided as part of the base TPF system, using TPF Internet server support, an HTTP server can be installed so that Web pages can be retrieved and TPF applications can be started from the Internet; for example, the Apache server.For more information see the Apache Web page at: http://www.apache.org

File Transfer Protocol (FTP) Server

The FTP server is used to transfer files between the TPF system and a remote host that supports Transmission Control Protocol/Internet Protocol (TCP/IP) and FTP clients. See *TPF Transmission Control Protocol/Internet Protocol* and *TPF Concepts and Structures* for more information on the FTP server.

Trivial File Transfer Protocol (TFTP) Server

The TPF system supports any TFTP client, such as AIX 3.1 and higher or DOS 5.0 and higher, that conforms to the Internet Activity Board (IAB) TFTP draft standard documented in Request for Comments (RFC) 1350.

However, the following differences are noted:

- The TFTP server creates files and directories specified in a TFTP write request that do not already exist.
- The TFTP server allows an existing file to be overwritten instead of issuing error 06 (file already exists).
- As data is received for a file being written, the TFTP server writes the data to a temporary file until the data transfer is complete. The temporary file is then renamed to the specified file.

  Overwriting of a file is controlled by its access permissions and is not done unless that file allows *other* write access permission.
- Access permissions for a new file are controlled by the TFTP configuration file.

  If the TFTP configuration file does not contain an AUTH directive, the access permissions default to 444 octal, which allows owner, group, and other read access. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the TFTP configuration file.
- The TFTP server provides a limited form of access control. In the TPF system, any path that is not explicitly allowed is not accessible.
- Logging of transmissions is available and controlled by the LOG directive in the TFTP configuration file. The following information is recorded:
  - The Internet Protocol (IP) address and port of the connecting system
  - The path and name of the file being transferred
  - The direction of transfer (read or write request)
  - The number of bytes transferred.

  If the TFTP configuration file does not contain a LOG directive, transmissions are logged to the `/tmp/tftp.log` file. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the TFTP configuration file.
- Client access to the TFTP server can be started and stopped using the ZINET START and ZINET STOP commnds, respectively. See *TPF Operations* for more information about the ZINET START and ZINET STOP commands.

Customizing the TFTP Server

The TFTP configuration file, `/etc/tftp.conf`, controls the behavior of the TFTP server for accessing files and the access permissions assigned to a file when it is stored. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the TFTP configuration file.

Security

The TFTP server does not perform any user validation. Carefully consider the file access rights allowed by read and write requests. The allow and deny directives that are specified in the TFTP configuration file are used to control file access. See *TPF Transmission Control Protocol/Internet Protocol* for more information about the TFTP configuration file.

File Names

All TPF file names specified by a TFTP client must be fully qualified path names that begin with a slash (/).

Using the TFTP Server from Another System

Use the tools for your system to create and maintain files of Web page content and transfer these files using the TFTP server to the TPF system where they can be accessed.

For more information about TFTP, see the following references.

- *Internet Architecture Board Standard 33, Request for Comments 1350*
- *AIX Version 4.1 Commands Reference*
- On DOS or DOS-based systems on a personal computer (PC), enter **HELP TFTP** or **TFTP ?**.

Syslog Daemon

The syslog daemon is a server process that provides a message logging facility for all application and system processes. The syslog daemon must be started before any other application or system process that uses it starts. Internet server applications and components use the syslog daemon for logging purposes and can also send trace information to the syslog daemon. Messages can be logged to files or to tape.

The syslog daemon processing is controlled by a configuration file named `/etc/syslog.conf` in which you define logging rules and output destinations for error messages, authorization violation messages, and trace data.

See *TPF Transmission Control Protocol/Internet Protocol* for more information about the syslog daemon, the configuration file, and application considerations.

TPF Internet Mail Servers

TPF Internet mail server support provides a set of servers that implement the standard Internet mail protocols on the TPF system. Users, or mail clients, interact with the TPF Internet mail servers to send and retrieve Internet mail, also known as electronic mail (e-mail). The TPF system supports the following standard Internet protocols:

- Simple Mail Transfer Protocol (SMTP)
- Internet Message Access Protocol (IMAP) Version 4
- Post Office Protocol (POP) Version 3.

SMTP describes how mail messages are delivered from one computer user to another. IMAP and POP describe how mail messages that are received on a computer (that is, a mail server) are retrieved by a mail client (usually another computer, such as a workstation).

See *TPF Transmission Control Protocol/Internet Protocol* for more information about TPF Internet mail server support.

Internet Server Application Considerations

An Internet server application must conform to one of the process models defined by the Internet daemon:

- WAIT

  The WAIT process model is for an Internet server application that is an iterative or single-thread program. The Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and does not start any more occurrences of the Internet server application until the one that is running ends.

  For TCP servers, the Internet daemon creates and monitors the listener socket. When a remote client connects, information about the socket that is associated with this client (the connected socket) is passed to the child process. The child process communicates with the remote client over the connected socket. If the child process ends and ***does not*** close the connected socket, the TPF system closes the connected socket automatically.

  For UDP servers, the Internet daemon creates and monitors the socket. When a message from a remote client is received, the socket is passed to the child process, allowing the child process to exchange data with the remote client.

- NOWAIT

  The NOWAIT process model is for an Internet server application that is a concurrent or multi-thread program. The Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and continues to start more occurrences of the Internet server application until a predefined limit is reached. When the limit is reached, no more occurrences are started until an occurrence that was previously started ends.

  For TCP servers, the Internet daemon creates and monitors the listener socket. When a remote client connects, information about the socket that is associated with this client (the connected socket) is passed to the child process. The child process communicates with the remote client over the connected socket. If the child process ends and ***does not*** close the connected socket, the TPF system closes the connected socket automatically. If you do not want the socket to be closed when the child process ends, the child process must issue an `fcntl` function with the O_TPF_NODDCLOSE option on the socket. You must do this when the child process passes the socket to another ECB (for example, using the `activate_on_receipt` function.)

  For UDP servers, the Internet daemon creates and monitors the socket. When a message from a remote client is received, the socket is passed to the child process, allowing the child process to exchange data with the remote client. The Internet daemon will not create another child process until the previous child process ends, or until the previous child process sends a SIGUSR1 signal to the Internet daemon. If child processes do not send SIGUSR1 signals, the NOWAIT model becomes single threaded and has the same characteristics as the WAIT model. If the UDP server application is bound to all local IP addresses and more than one local IP address exists, the child process should not send the SIGUSR1 signal until after the process has sent all its data to the remote client. If a SIGUSR1 signal is sent before all the data is sent, the wrong local IP address can be placed in the packets sent to the remote client, causing that data to be discarded.

- AOR

  The AOR process model is for an Internet server application that is a concurrent or multithread program. The AOR process model can be used only for TCP servers, not UDP. The Internet daemon creates and monitors the listener socket.

When a remote client connects, the Internet daemon issues an `activate_on_receipt` or `activate_on_receipt_with_length` function to pass control of the new socket to your TCP server application when the first message is received from the remote client.

- DAEMON

  The DAEMON process model is for an Internet server application that is started and monitored by the Internet daemon. The Internet daemon starts the Internet server application as a child process by using the `tpf_fork` function and does not start any more occurrences of the Internet server application until the one that is running ends. The Internet daemon does not create or monitor any sockets for this process model.
- NOLISTEN

  The NOLISTEN process model is for an Internet server application that the Internet daemon only starts. The Internet daemon does not create or monitor any sockets, nor does it monitor the server application.
- RPC

  The RPC process model is for a remote procedure call (RPC) server application that the Internet daemon only starts. The Internet daemon does not create or monitor any sockets, nor does it monitor the server application.

Because these process models provide different levels of control, the overhead of system resource can be affected. The AOR process model has the potential to use system resources more effectively than the WAIT and NOWAIT process models. The AOR process models uses the `activate_on_receipt` or `activate_on_receipt_with_length` function to create the entry control block (ECB) for the Internet server application only after data is received on a connected socket, whereas the WAIT and NOWAIT process models use the `tpf_fork` function to create an ECB for immediate processing, which requires the Internet server application to request data and then wait for the data to arrive.

For the NOWAIT process model, use the MAXPROC parameter in the ZINET ADD or ZINET ALTER command to limit the number of occurrences of an Internet server application that the Internet daemon starts. For the AOR process model, there is no throttling control, so the Internet daemon can potentially start more occurrences of the Internet server application until system resources are depleted. See *TPF Operations* for more information about the ZINET ADD and ZINET ALTER commands.

In general, a ported Internet server application uses the WAIT or NOWAIT process model and an Internet server application designed specifically for TPF architecture uses the AOR process model.

When the Internet daemon is running, there is one long-running ECB for the Internet daemon monitor program and one long-running ECB for each server application that the Internet daemon is monitoring. Some E-type loader operations are not completed until all old ECBs exit in the system. When the Internet daemon detects that the system activation number has changed, the Internet daemon recycles itself. A new instance of the Internet daemon is immediately created in new ECBs, and the old Internet daemon ECBs exit when all their child processes end. When the Internet daemon recycles, the effect on the server application and sockets is based on the process model:

- For the WAIT, NOWAIT, and AOR process models, there is no disruption to the sockets. The fact that the Internet daemon has recycled itself is completely transparent to the server and remote client applications.

- For the NOLISTEN and RPC process models, no action is taken when the Internet daemon recycles. Servers using these models are activated when the Internet daemon first starts and, after that, the Internet daemon does not monitor those applications.
- For the DAEMON process model, the Internet daemon issues a `kill` function to stop the old application server instance. The new instance of the Internet daemon starts a new server application instance as soon as the old server instance ends. The Internet daemon does not create or monitor sockets for this process model; therefore, your application must take the appropriate actions regarding any sockets that it created. A common implementation is for the old server application to close the sockets and have the new server application instance start new sockets. Another method is to keep the sockets active by having the old server application instance save the socket descriptors in a user table that the new server application instance picks up and uses.

When you enter the ZINET STOP command to stop a server application, the actions that are taken are based on the process model:

- For a TCP server using the WAIT, NOWAIT, or AOR process model, the listener socket is closed and no new connections can be started with this server. Existing connections (sockets) between remote clients and this server are allowed to continue their normal processing. If you want to break the existing connections that use TCP/IP native stack support, enter **ZSOCK INACT LPORT-***lport*, where *lport* is the port number of the TCP server application.
- For a UDP server using the WAIT or NOWAIT process model, the socket is closed.
- For the NOLISTEN and RPC process models, no action is taken because the Internet daemon does not monitor these process models.
- For the DAEMON process model, the Internet daemon issues a `kill` function to stop the application server. When using this process model, ensure that your Internet server application uses the `signal` function to enable the SIGTERM signal.

See *TPF Operations* for more information about the ZINET STOP and ZSOCK commands.

Process Models

The following sections describe the processing flow for each process model based on the transport protocol (Transmission Control Protocol (TCP) or User Datagram Protocol (UDP)).

TCP and the NOWAIT Process Model

In a TCP environment, if you specify the NOWAIT process model, the Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and then immediately processes the next request. Figure 26 on page 179 shows the relationship of the parent and child processes and an overview of the logic flow of process control and socket APIs.

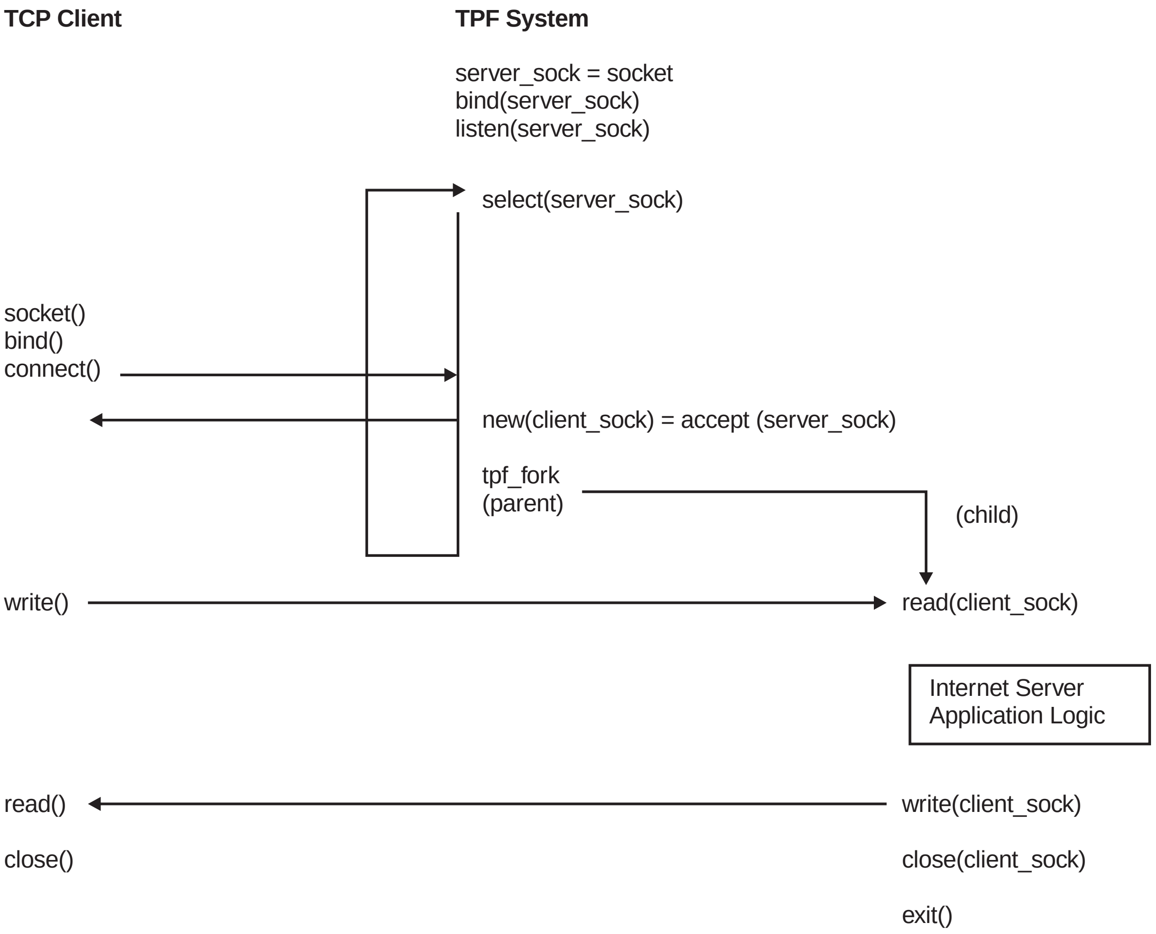


Figure 26. TCP and the NOWAIT Process Model. After starting the Internet server application, the Internet daemon processes the next request from the Internet.

TCP and the WAIT Process Model

In a TCP environment, if you specify the WAIT process model, the Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and does not process the next request (that is, the Internet daemon waits) until the Internet server application ends and the TPF system sends a SIGCHLD signal. Figure 27 on page 180 shows the relationship of the parent and child processes and an overview of the logic flow of process control and socket APIs.

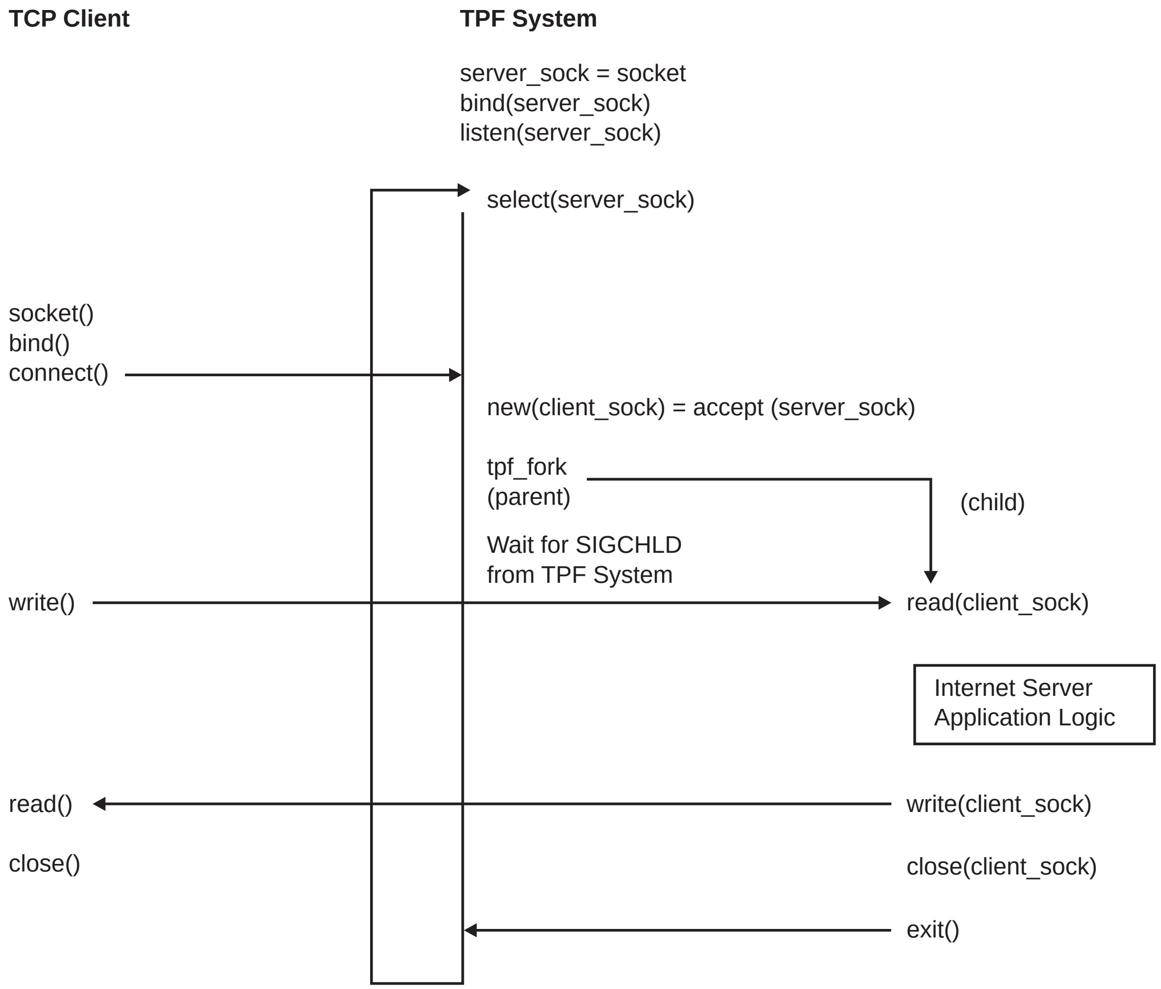


Figure 27. TCP and the WAIT Process Model. After starting the Internet server application (child process), the Internet daemon (parent process) does not continue processing until the Internet server application ends and the TPF system signals to the parent process that the child process has ended.

TCP and the AOR Process Model

In a TCP environment, if you specify the AOR process model, the Internet server application is started as a new entry control block (ECB) when there is data for it by an `activate_on_receipt` function call. Meanwhile, the Internet daemon continues to process requests from the Internet. Figure 28 on page 181 shows the relationship of the Internet daemon listener and an Internet server application using the AOR process model and an overview of the logic flow of socket APIs.

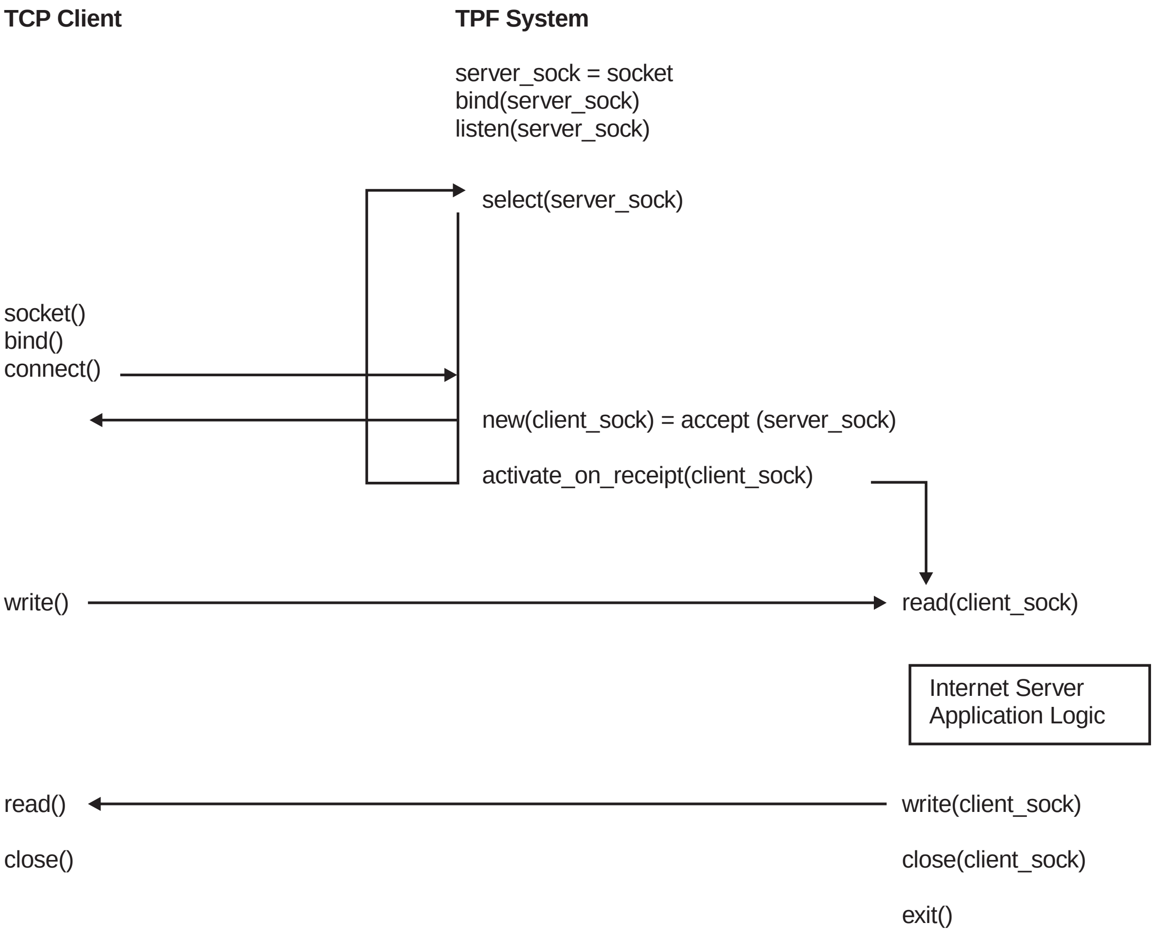


Figure 28. TCP and the AOR Process Model. After starting the Internet server application, the Internet daemon accepts the next request from the Internet.

TCP and the DAEMON Process Model

In a TCP environment, if you specify the DAEMON process model, the Internet daemon starts the Internet server application as a child process by using the `tpf_fork` function and does not process the next request (that is, the Internet daemon waits) until the Internet server application ends and the TPF system sends a SIGCHLD signal back to the Internet daemon. Figure 29 on page 182 shows the relationship of the parent and child processes and an overview of the logic flow of process control and socket APIs.

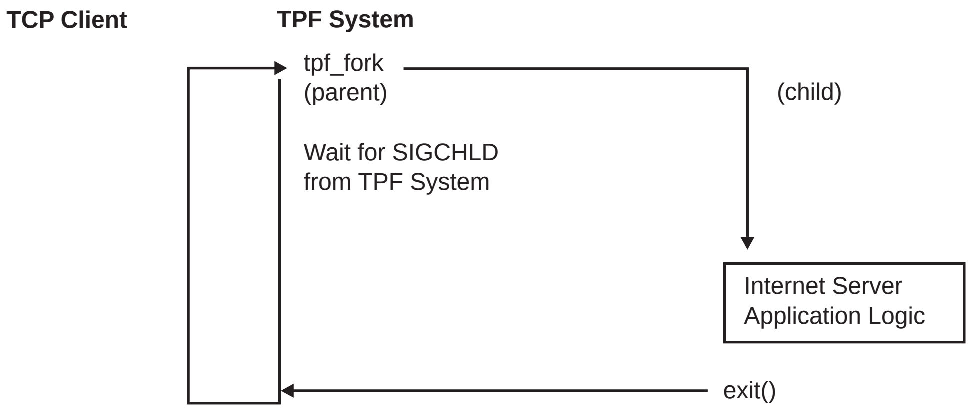


Figure 29. TCP and the DAEMON Process Model. After starting the Internet server application (child process), the Internet daemon (parent process) does not continue processing until the Internet server application ends and the TPF system signals to the parent process that the child process has ended.

UDP and the NOWAIT Process Model

In a UDP environment, if you specify the NOWAIT process model, the Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and then waits until the child process sends a SIGUSR1 signal before continuing its processing. Figure 30 on page 183 shows the relationship of the parent and child processes and an overview of the logic flow of process control and socket APIs.

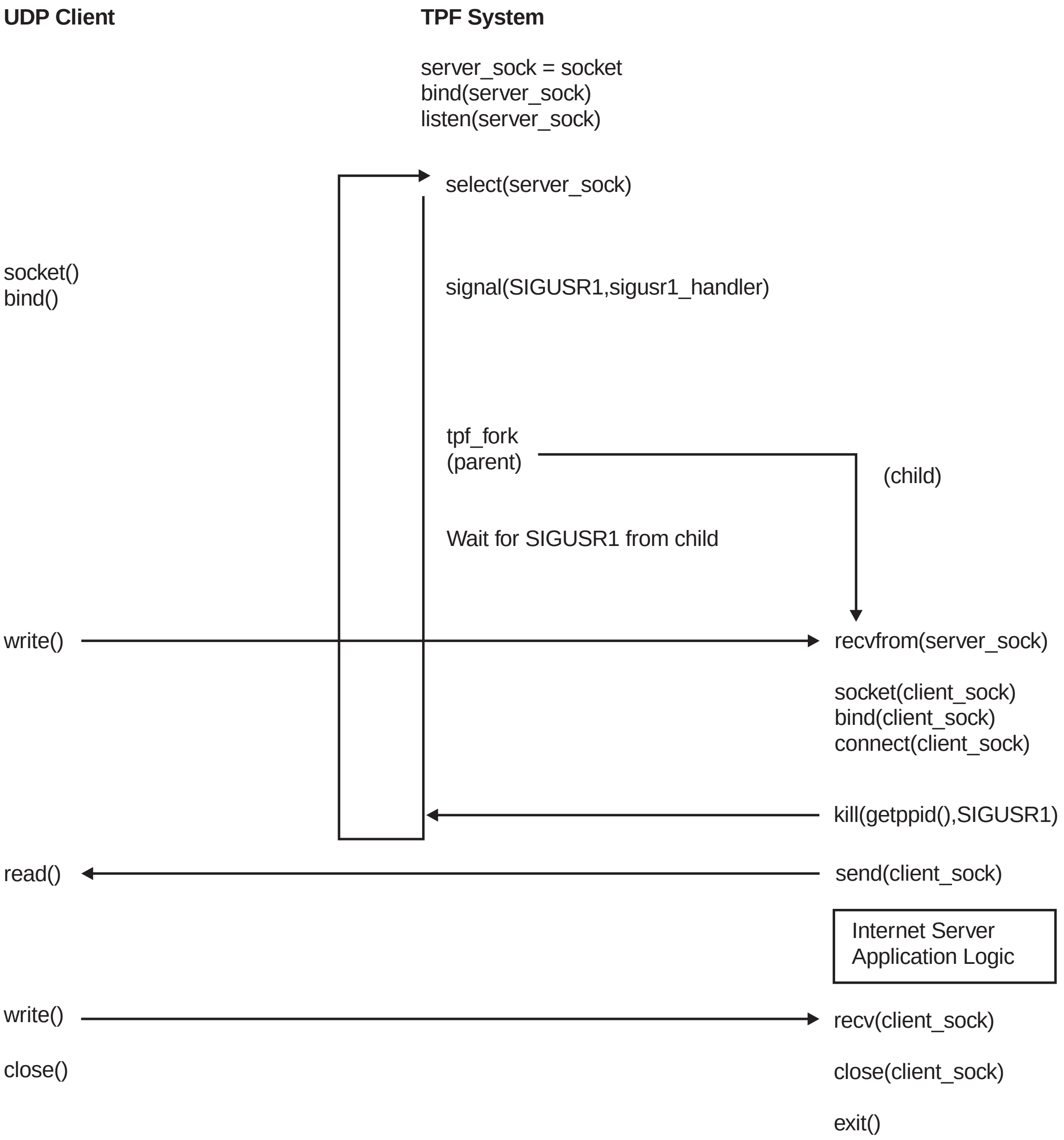


Figure 30. UDP and the NOWAIT Process Model. After starting the Internet server application (child process), the Internet daemon (parent process) waits until the child process sends a SIGUSR1 signal before continuing its processing.

UDP and the WAIT Process Model

In a UDP environment, if you specify the WAIT process model, the Internet daemon starts the Internet server application as a child process using the `tpf_fork` function and does not process the next request (that is, the Internet daemon waits) until the Internet server application ends and the TPF system sends a SIGCHLD signal. Figure 31 on page 184 shows the relationship of the parent and child processes and an overview of the logic flow of process control and socket APIs.

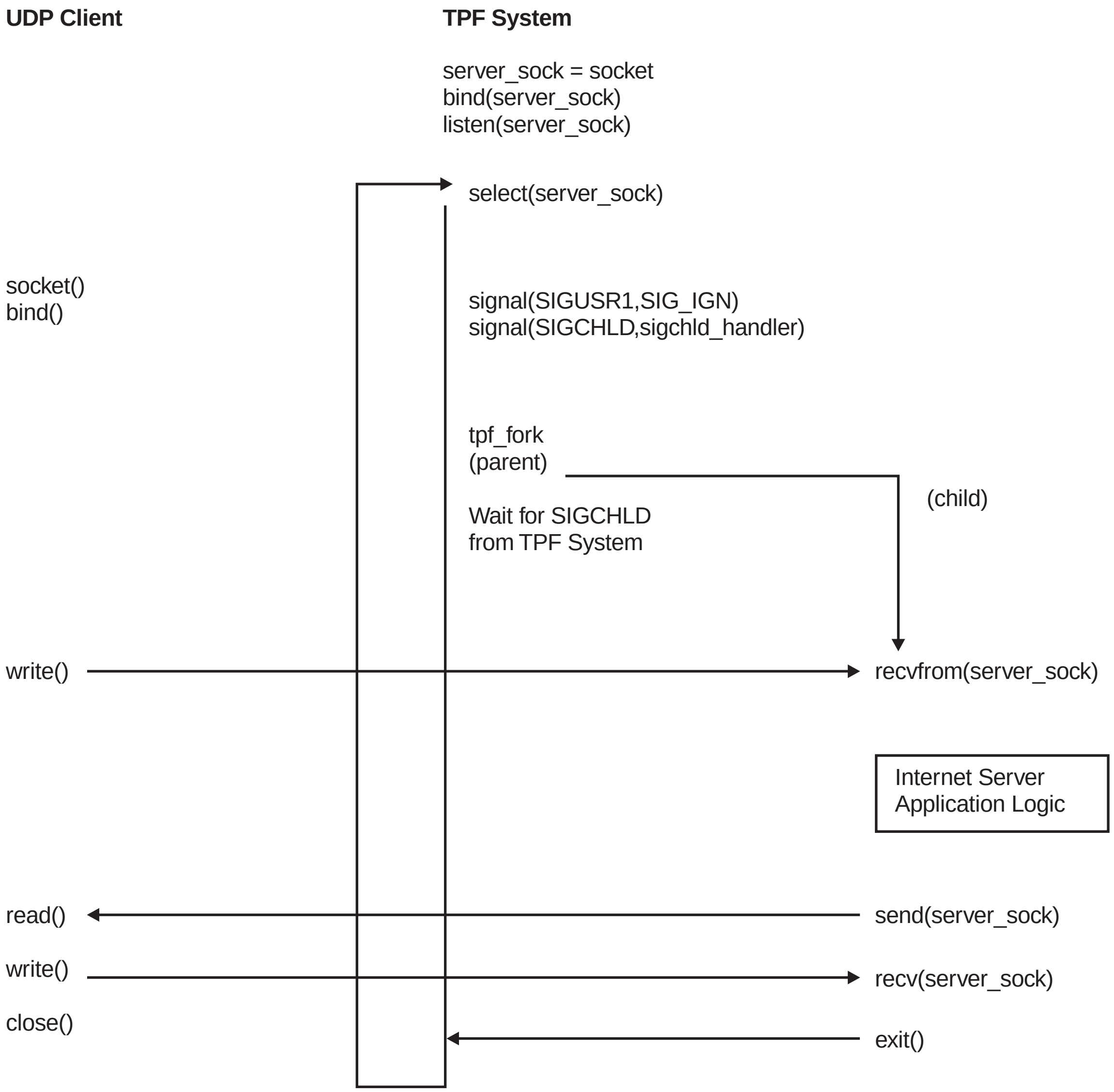


Figure 31. UDP and the WAIT Process Model. The Internet daemon (parent process) does not continue processing until the Internet server application (child process) ends and the TPF system signals to the parent process that the child process has ended.

Internet Server Application Interface

The following sections describe the interface between the Internet daemon and an Internet server application for the process model based on the transport protocol (TCP or UDP).

TCP and the WAIT Process Model

The Internet server application is started using the `tpf_fork` function and must be a C or C++ `main` program.

When the Internet server application receives control, the following data is in the entry control block (ECB):

EBW000–003
Socket descriptor, defined as (long int).

The Internet daemon has accepted a socket connection, and the socket descriptor passed in EBW000 is for the connected socket.

EBW004–011
1- to 8-character alphanumeric parameter string defined for the application in the IDCF; this parameter string corresponds to the PARM parameter in the ZINET ADD command.

EBW012–021
Server name; this corresponds to the SERVER parameter in the ZINET ADD command.

Although it is not necessary for the Internet server application to explicitly reference the associated process block, the following inheritance properties, as defined by UNIX, are passed to the Internet server application:

File descriptor 0
Socket descriptor for standard input (`stdin`).

File descriptor 1
Socket descriptor for standard output (`stdout`).

Real and effective user and group IDs
This information is obtained from the password file based on the Internet daemon configuration file (IDCF) entry that was initialized by the USER parameter in the ZINET ADD command.

Working directory
This information is obtained from the password file based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

The Internet daemon does not accept another connection or data until the ECB for the Internet server application ends.

The Internet server application must close the socket from which the request was received.

See *TPF Operations* for more information about the ZINET ADD command.

TCP and the NOWAIT Process Model

The Internet server application is started using the `tpf_fork` function and must be a C or C++ `main` program.

When the Internet server application receives control, the following data is in the entry control block (ECB):

EBW000–003
Socket descriptor, defined as (long int).

The Internet daemon has accepted a socket connection, and the socket descriptor is passed in EBW000 for the connected socket.

EBW004–011
1- to 8-character alphanumeric parameter string defined for the application in the IDCF; this parameter string corresponds to the PARM parameter in the ZINET ADD command.

EBW012–021
Server name; this corresponds to the SERVER parameter in the ZINET ADD command.

Although it is not necessary for the Internet server application to explicitly reference the associated process block, the following inheritance properties, as defined by UNIX, are passed to the Internet server application:

File descriptor 0
Socket descriptor for standard input (`stdin`).

File descriptor 1
Socket descriptor for standard output (`stdout`).

Real and effective user and group IDs
This information is obtained from the password and group files based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

Working directory
This information is obtained from the password file based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

The Internet server application must close the socket from which the request was received.

See *TPF Operations* for more information about the ZINET ADD command.

UDP and the WAIT Process Model

The Internet server application is started using the `tpf_fork` function and must be a C or C++ `main` program.

When the Internet server application receives control, the following data is in the entry control block (ECB):

EBW000–003
Socket descriptor, defined as (long int).

EBW004–011
1- to 8-character alphanumeric parameter string defined for the application in the IDCF; this parameter string corresponds to the PARM parameter in the ZINET ADD command.

EBW012–021
Server name; this corresponds to the SERVER parameter in the ZINET ADD command.

Although it is not necessary for the Internet server application to explicitly reference the associated process block, the following inheritance properties, as defined by UNIX, are passed to the Internet server application:

File descriptor 0
Socket descriptor for standard input (`stdin`).

File descriptor 1
Socket descriptor for standard output (`stdout`).

Real and effective user and group IDs
This information is obtained from the password and group files based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

Working directory
This information is obtained from the password file based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

The Internet daemon does not accept another connection or data until the ECB for the Internet server application ends.

The Internet server application must not close the socket from which the request was received.

See *TPF Operations* for more information about the ZINET ADD command.

UDP and the NOWAIT Process Model

The Internet server application is started using the `tpf_fork` function and must be a C or C++ `main` program.

When the Internet server application receives control, the following data is in the entry control block (ECB):

EBW000–003
Socket descriptor, defined as (long int).

EBW004–011
1- to 8-character alphanumeric parameter string defined for the application in the IDCF; this parameter string corresponds to the PARM parameter in the ZINET ADD command.

EBW012–021
Server name; this corresponds to the SERVER parameter in the ZINET ADD command.

Although it is not necessary for the Internet server application to explicitly reference the associated process block, the following inheritance properties, as defined by UNIX, are passed to the Internet server application:

File descriptor 0
Socket descriptor for standard input (`stdin`).

File descriptor 1
Socket descriptor for standard output (`stdout`).

Real and effective user and group IDs
This information is obtained from the password and group files based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

Working directory
This information is obtained from the password file based on the IDCF entry that was initialized by the USER parameter in the ZINET ADD command.

The Internet server application must not close the socket from which the request was received.

See *TPF Operations* for more information about the ZINET ADD command.

TCP and the AOR Process Model

The Internet daemon uses the `activate_on_receipt` function to start the Internet server application.

When the Internet server application receives control, most of the data in the ECB is provided by the TPF system. However, the Internet daemon obtained the user parameter string from the IDCF that was initialized by the PARM parameter in the ZINET ADD command. This parameter string is the data at EBW004–011.

See *TPF Operations* for more information about the ZINET ADD command. See the description of the `activate_on_receipt` function in *TPF Transmission Control Protocol/Internet Protocol* for the layout of the data in the ECB.

The Internet daemon issues the `activate_on_receipt` function for a connected socket.

There are no inheritance properties associated with this model because the Internet server application is started as a new ECB rather than a child process.

Add an Internet Server Application to the IDCF

Use the ZINET ADD command to add an Internet server application to the IDCF. See *TPF Transmission Control Protocol/Internet Protocol* and *TPF Operations* for more information about the ZINET ADD command and adding entries to the IDCF.

Considerations for Using the Internet Daemon to Start a TPF Program

The NOLISTEN process model in the Internet daemon is provided to start any TPF application that does not communicate over the Internet.

Interface

A TPF program is started using the `swisc_create` function.

When the application receives control, the following data is in the entry control block (ECB):

EBW000–003
4 bytes of zeros (X'00').

EBW008–011
1- to 8-character alphanumeric parameter string defined for the application in the IDCF; this parameter string corresponds to the PARM parameter in the ZINET ADD command.

EBW012–021
Server name; this corresponds to the SERVER parameter in the ZINET ADD command.

There are no inheritance properties associated with this model because the TPF program is started as a new ECB rather than a child process.

See *TPF Operations* for more information about the ZINET ADD command.

Starting a TPF Application from the Internet

Assuming that a Hypertext Transfer Protocol (HTTP) server is installed, TPF Internet server support provides a way to start an E-type program based on an application name as described in the HTTP request. So, it appears that starting an application is similar to the way it is implemented on a UNIX system while, in reality, the program is started using the TPF system loader.

On another system, write an executable script, which is a type of executable file, and transfer it to the TPF system using the TFTP server. The TFTP server has the capability to convert the data to EBCDIC if necessary.

When the HTTP server processes the executable script, it creates a child process that is associated with an E-type program in the TPF application using the `tpf_fork` function. The HTTP server must have execute access permission to the executable script.

A TPF application can direct its output to the Internet using file system APIs on file descriptors that refer to sockets.

Executable Script

An *executable script* is a type of executable file that can be used by the `tpf_fork` function to start a TPF application.

The content of an executable script is EBCDIC text of the form:

#!*interpreter name*

The *interpreter name* is the 4-character loader segment (E-type program) name containing a `main` function.

Example

Using the example of asking for the availability of an airline flight, assume the first program in the availability application is QZZ2. The associated executable scripts would contain:

#!QZZ2

For this example, when the client on the Internet requests availability, the Internet daemon starts the HTTP server, which creates the child process that runs E-type program QZZ2.

Understanding TPF Remote Procedure Call

Remote procedure call (RPC) allows applications on one workstation to call functions that reside on and are run by another workstation. Figure 32 shows how RPC can be used to call various types of new and existing applications. The requesting application is not concerned with networking issues and data representation of passed parameters between the two workstations because these are resolved when the RPC client and server applications are developed and the RPC interface is defined. The interface consists of the supported functions and the format of their parameters, and is created by using the Interface Definition Language (IDL). Each RPC interface requires a universal unique identifier (UUID) that is created by using a UUID generator utility. TPF RPC requires that all the offline utilities, such as the UUID generator and the IDL compiler, are run on the OS/390 system.

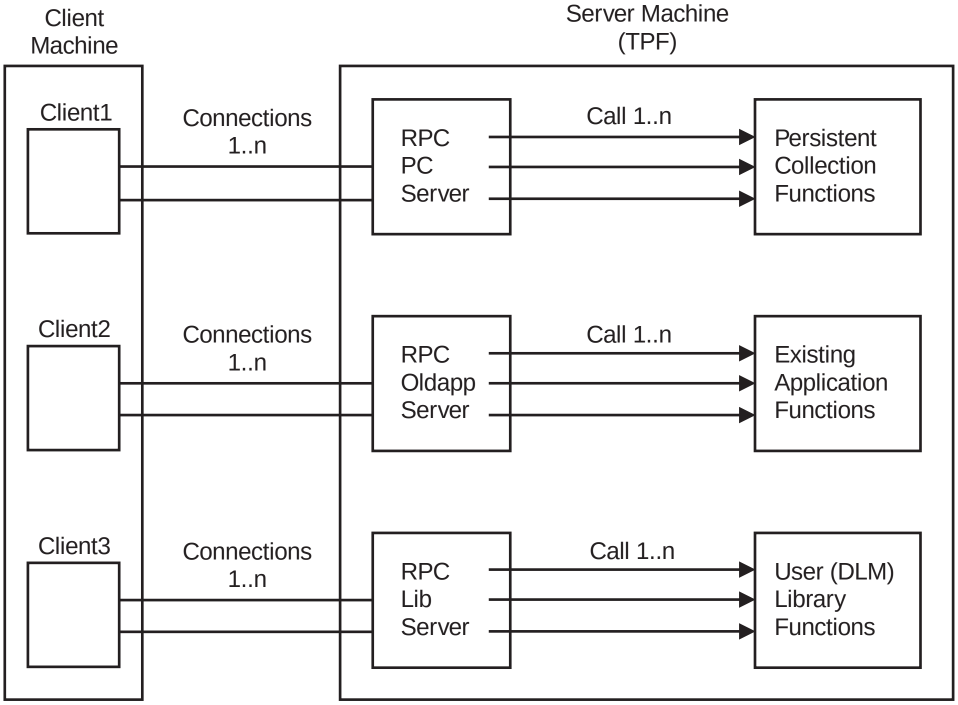


Figure 32. RPC Servers Used to Call Applications

Interface Definition Language and Stub Files

Client and server code are tied together through the use of the interface definition. The interface definition describes the set of procedures that is offered by the interface. The interface definition file is coded using the Interface Definition Language (IDL).

To create an RPC interface, you must use the IDL to define each RPC function and the format of the input and output parameters. An IDL compiler on both the client and server platforms is used to compile a *.idl* file, and also to generate header files and *stub* (C source) files. The header files are included and the stub files are linked in both the client and server application code; this code includes the required functions to convert between client and server data formats and to handle network communications between the client and the server.

TPF Modifications to Distributed Computing Environment (DCE) RPC

Client applications running on any IBM or non-IBM DCE platform are able to run remote procedure calls to a TPF server. All DCE services are available to client applications; however, the TPF 4.1 system supports only a subset of the DCE RPC services. The following are elements of a traditional DCE RPC environment that can be used with the TPF implementation:

- Server support only (no client support)
- Unauthenticated RPC only (no DCE security service or Kerberos provided)
- An offline process to load TPF server information into a directory server (no directory service application programming interfaces (APIs))
- Preassigned server port numbers (no dynamic server port assignment at run time). Client applications must obtain full binding information about the TPF server.

Creating an RPC Interface for TPF

Figure 33 shows how an RPC interface is created and used in the RPC client/server environment.

Note: The directory server is not needed if string bindings are used that contain the complete binding information.

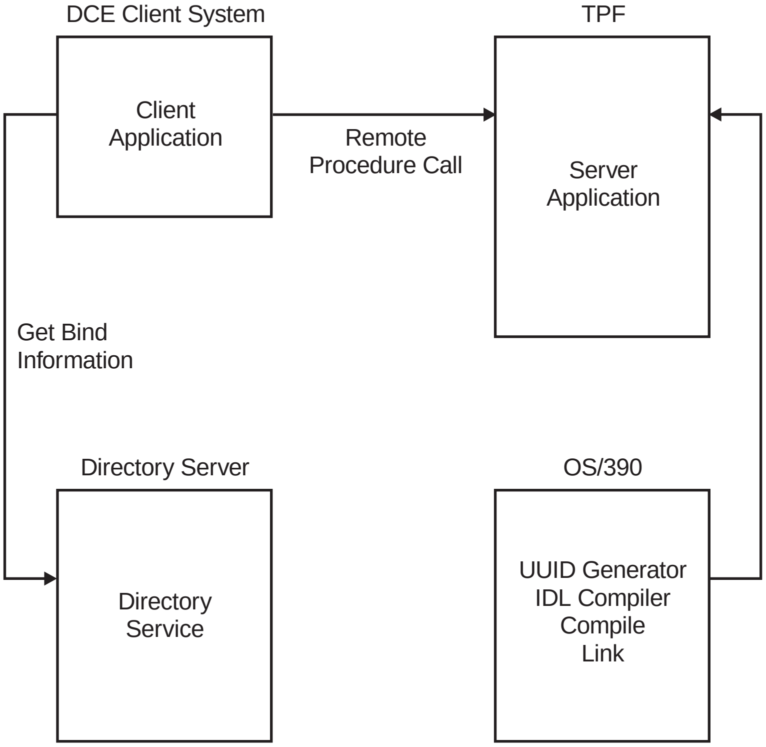


Figure 33. Remote Procedure Call Overview for Client and Server Platforms

To create an RPC client/server application, do the following:

1. Generate a universal unique identifier (UUID) for the new RPC interface by using the UUID Generator Utility on an OS/390 system.
2. Create an Interface Definition Language (IDL) file that includes the UUID and the remote procedures with their input and output parameters.

3. Compile the IDL file with an IDL compiler to generate header files and stub files for client and server applications on an OS/390 system.
4. Include the header file in the server application. Compile and link the server stub file and the server application on the server platform (this would be where you compile and link all your TPF code).
5. Include the header file in the client application. Compile and link the client stub file and the client application on the client platform.
6. Load the server code to your TPF 4.1 system. RPC servers are processor and subsystem unique.
7. Cycle the TPF 4.1 system to 1052 state or higher.
8. To increase thread resources, change the thread parameters in keypoint A (CTKA) by entering the ZCTKA ALTER command with the MTHD and TSTK parameters specified. Restart your system by entering the ZRIPL command.
9. Define an Internet daemon (INETD) entry for the new server by entering the ZINET ADD command with the S, MODEL-RPC, and PGM parameters specified.
10. Cycle the TPF 4.1 system to CRAS state or higher.
11. Verify that TCP/IP support is active in the TPF 4.1 system and that required offload devices or Internet Protocol (IP) routers are available and active. Enter the ZCLAW DISPLAY command with the ACTIVE parameter specified to check the active CLAW workstations, and the ZTTCP DISPLAY command with the ACTIVE parameter specified to display all active IP routers.
12. Start the server using the INETD by entering the ZINET START command with the S parameter specified.

Clients can now access remote procedures from any client platform.

See *TPF Operations* for more information about the ZCLAW DISPLAY, ZCTKA ALTER, ZINET ADD, ZINET START, ZRIPL, and ZTTCP DISPLAY commands.

TPF RPC Run-Time Library

The Open Software Foundation Distributed Computing Environment (OSF DCE) RPC run-time library consists of RPC routines that perform a variety of functions. The TPF 4.1 system has implemented the subset of those routines that relate to server support. The RPC run-time library was ported from MVS/ESA OpenEdition DCE. The code is written in C language and is implemented as a dynamic link library (DLL) on the TPF 4.1 system; therefore, TPF RPC server applications must be compiled with the DLL option.

The RPC run-time library allows you to develop RPC server applications that are accessed using Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). The RPC library name is CRPC and the library ordinal number is 0011. The RPC library APIs establish all required client/server connections using socket APIs. The following is a list of supported RPC run-time library APIs:

`rpc_binding_copy`	Returns a copy of a binding handle.
`rpc_binding_free`	Releases binding handle resources.
`rpc_binding_inq_object`	Returns the object UUID from a binding handle.
`rpc_binding_to_string_binding`	Returns a string representation of a binding handle.

rpc_binding_vector_free	Frees the memory used to store a vector of binding handles.
rpc_if_id_vector_free	Frees a vector and the interface identifier structure it contains.
rpc_if_inq_id	Returns the interface identifier for an interface specification.
rpc_mgmt_inq_if_ids	Returns a vector of interface identifiers of interfaces a server offers.
rpc_mgmt_inq_stats	Returns RPC run-time library statistics.
rpc_mgmt_is_server_listening	Tells whether a server is listening for remote procedure calls.
rpc_mgmt_stats_vector_free	Frees a statistics vector.
rpc_mgmt_stop_server_listening	Tells a server to stop listening for remote procedure calls.
rpc_network_inq_protseqs	Returns all protocol sequences supported by both the RPC run-time library and the operating system.
rpc_network_is_protseq_valid	Tells whether the specified protocol sequence is supported by both the RPC run-time library and the operating system.
rpc_object_inq_type	Returns the type of an object.
rpc_object_set_inq_fn	Registers an object inquiry function.
rpc_object_set_type	Registers the type of an object with the RPC run-time library.
rpc_protseq_vector_free	Frees the memory used by a vector and its protocol sequences.
rpc_server_inq_bindings	Returns binding handles for communications with a server.
rpc_server_inq_if	Returns the manager entry point vector registered for an interface.
rpc_server_listen	Tells the RPC run-time library to listen for remote procedure calls.
rpc_server_register_if	Registers an interface with the RPC run-time library.
rpc_server_unregister_if	Removes an interface from the RPC run-time library.
rpc_server_use_all_protseqs_if	Tells the RPC run-time library to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls.
rpc_server_use_protseq_ep	Tells the RPC run-time library to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.
rpc_server_use_protseq_if	Tells the RPC run-time library to use the specified

	protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls.
rpc_ss_allocate	Allocates memory within the RPC stub memory management scheme.
rpc_ss_free	Frees memory allocated by the `rpc_ss_allocate` routine.
rpc_string_binding_compose	Combines the components of a string binding into a string binding.
rpc_string_binding_parse	Returns, as separate strings, the components of a string binding.
rpc_string_free	Frees a character string allocated by the run-time library.
uuid_compare	Compares two UUIDs and determines their order.
uuid_create	Creates a new UUID.
uuid_create_nil	Creates a nil UUID.
uuid_equal	Determines if two UUIDs are equal.
uuid_from_string	Converts a string UUID to its binary representation.
uuid_hash	Creates a hash value for a UUID.
uuid_is_nil	Determines if a UUID is nil.
uuid_to_string	Converts a UUID from a binary representation to a string representation.

See the *OSF DCE Application Development Reference* for more information on the DCE RPC APIs.

The first time an RPC API is called, the run-time library is initialized. After the first API is completed, the server runs in a thread environment. See Threads on page 196 for more information about the thread environment and thread safety.

RPC Calls

You can use the `rpc_server_listen` API with the `max_calls_exec` parameter to specify the maximum number of RPC server calls that are running concurrently on the TPF 4.1 system. The run-time library accepts or rejects an RPC call (the call is not queued) when accepting a connection for TCP implementation; this is determined by the number of calls that are currently running. The `max_calls_exec` value must take into account thread resources and must be a number greater than zero and less than the maximum number of threads per process. You can use the following formula as a guideline:

```
(2 * max_call) + 3 threads defined
```

To change the maximum number of threads allowed for each process, see the information for the MTHD parameter of the ZCTKA ALTER command in *TPF Operations*.

If the TPF 4.1 system cannot acquire heap storage while attempting to receive a call, a dump occurs and the RPC server is exited. If the TPF 4.1 system cannot create a thread to process the RPC call (because of maximum thread limitations), RPC closes the newly accepted socket and the server continues to run. Done at accept time, this form of *throttling* rejects the RPC calls if system resources are not available and processes the next RPC call as soon as system resources become available. Although it has an impact on the client, this throttling is not unique to the TPF 4.1 system. If errors such as queue limits exceeded or network failures occur, the client can retry the call.

Threads

OSF DCE RPC is built on top of a thread model. In the TPF 4.1 system, the thread library name is CTHD and the library ordinal number is 0008.

The Thread Environment

To enable your TPF 4.1 system for threads, enter the ZCTKA ALTER command and specify the MTHD and TSTK parameters. The MTHD parameter specifies the maximum number of threads allowed in a process. The TSTK parameter specifies the maximum number of 4-KB ISO-C stack frames for each thread. The TSTK parameter must be a value from 4 to 1024 and must be a power of 2. See *TPF Operations* for more information on the ZCTKA ALTER command.

Thread Safety

Server applications run as threads and, therefore, must be *thread safe*. Most TPF services are currently thread safe; however, some services, such as file system and C function trace are not. Any function that causes the file system to be initialized, such as `printf`, and is used by a threaded application will cause results that cannot be predicted in the threaded application, such as depletion of 4-KB frames. C function trace is skipped in a threaded environment. The TPF recommendation is for RPC server applications to exit the thread environment to run the call (see `tpf_cresc` in the *TPF C/C++ Language Support User's Guide* as one way to exit the thread environment), and then return to the thread environment when the call is completed. Parameters can still be passed back and forth using this method.

In addition to the server applications, the server itself runs as a thread. Typically, the server contains RPC run-time APIs and additional processing, such as initialization and cleanup. You must be certain that any additional processing that occurs in the server is thread safe.

RPC Servers

RPC servers are subsystem unique.

Note: Because the TPF 4.1 system does not support *dynamic endpoints (port numbers)* for RPC servers, the same RPC server cannot run in multiple subsystems unless there is a unique connection for the server.

The local IP address or port number has to be unique to establish a unique connection.

Whenever a remote procedure call takes a system error and the ECB exits, the RPC server is deactivated and cleaned up. The TPF 4.1 system provides the `tpf_RPC_options` API with a `TPF_RPC_OPTIONS_EXIT_THREAD` parameter so you can avoid this server deactivation. You can use this parameter to make the issuing thread ECB exit without stopping its process. The `TPF_RPC_OPTIONS_EXIT_THREAD` parameter must be issued in the call thread immediately before the requested RPC interface function is run. The `tpf_RPC_options` API provides a corresponding reset parameter, `TPF_RPC_OPTIONS_EXIT_PROCESS`, that must be issued before the remote procedure call returns.

Starting and Stopping RPC Servers

You can start or stop RPC servers through the following:

- Internet daemon (INETD), by using the ZINET ADD, ZINET ALTER, ZINET START, and ZINET STOP commands
- Cycle-up and cycle-down processing
- E-type loader, by using the ZOLDR ACTIVATE, ZOLDR DEACTIVATE, and ZOLDR EXCLUDE commands.

INETD

You can add, change, start, or stop RPC servers by using the following commands for the INETD:

- ZINET ADD adds an RPC server entry to the Internet daemon configuration file. You must specify the MODEL=RPC parameter so that INETD will only start and stop the TPF RPC server and not manage it. TPF RPC servers use *static binding* (the IP address and port number are predetermined), so the IP and PORT parameters have no value for RPC.
- ZINET ALTER allows you to change the server parameters and the type of activation.
- ZINET START starts a specified server.
- ZINET STOP stops a specified server.

  When you enter the ZINET STOP command, the INETD marks the specified server as inactive. The RPC run-time library detects that the ZINET STOP command was issued and attempts an orderly shutdown. The server stops listening on the port and handles all queued or active requests before cleaning up. Two messages are displayed for this condition. INETD processing indicates that the server was stopped (as a direct response to the ZINET STOP command that was entered). The RPC run-time library then displays a message indicating that the server is shut down.

See *TPF Operations* for more information about the ZINET ADD, ZINET ALTER, ZINET START, and ZINET STOP commands.

TPF Cycle-Up and Cycle-Down Processing

RPC servers can be started and stopped automatically in cycle-up and cycle-down processing. This support is implemented through the INETD.

Note: Transmission Control Protocol/Internet Protocol (TCP/IP) and socket support deactivates connections as part of cycle-down processing. Any or all of the outstanding RPC requests may or may not be processed during cycle-down processing. RPC takes a communication error cycling to 1052 state; however, RPC will end normally when cycling down to CRAS state (if the INETD is not set to be active in CRAS state). See the ZINET commands in *TPF Operations* for more information.

E-Type Loader

RPC servers are automatically recycled whenever any loadset is activated, deactivated, or excluded by the E-type loader. Continuous service is provided between the time that the server is shut down and the server is reactivated. Any remote procedure calls received during this time are queued and handled by the newly activated server. All remote procedure calls in the TPF 4.1 system cause an

ECB to be created that inherits the system activation number. Newly activated functions will not be used by the RPC server until the server is restarted and can create ECBs with the new system activation number.

The `tpf_is_RPCServer_auto_restarted` API is provided so you can have a server application query if it is automatically restarted. Applications can skip certain initial startup code for a server that is restarted.

Performance and Tuning for RPC

Three fields have been added to the System Summary Report:

- To ensure that there are enough threads defined to the TPF 4.1 system for any application that uses threads, data is collected from the following fields:
  - The maximum number of threads for each process

    The maximum number of threads active in any process at any time during an iteration of data collection.
  - The high-water mark number of threads active in a process

    The maximum number of threads active in any process at any point in time during the current initial program load (IPL).

  If a thread application (for example, RPC) is using a large number of threads compared to the number defined in keypoint A (CTKA), you can enter the ZCTKA ALTER command to change the maximum number of threads.
- The third field is the maximum number of frames on the frames pending list, which is collected to help determine if additional 4-KB frames are needed. If the maximum number is greater than 10% of the frames in the TPF 4.1 system, a system shutdown can occur because it is low on available frames.

  Frames that are released by threaded ECBs are placed on the frames pending list until it is safe to reuse them; for example, after a purge of the translation look-aside buffer (PTLB) is performed on all I-streams. If a large number of frames remains on the frames pending list, additional 4-KB frames should be generated in the TPF 4.1 system. See *TPF System Generation* for more information about the CORREQ macro.

RPC Storage Considerations

RPC servers run in a threaded environment. You need to consider both the size of the ECB heap area and the maximum number of threads in a process. As each thread is created in a process, the heap area of the initial thread is shared with the new thread. The maximum number of threads in a process affects the size of the collective heap.

You can enter the ZCTKA ALTER command to modify heap storage values. The maximum size of the ECB heap is set by specifying the EMPS parameter. The maximum number of 4-KB frames that an ECB can acquire for heap storage is set by specifying the MMHS parameter. Specifying these parameters affects all ECBs in the TPF 4.1 system. In a threaded environment, the value in the MMHS parameter may be too small to accommodate the collective heap. You can modify the CE2MPF field in the ECB to override the value in the MMHS parameter; this will allow for the number of 4-KB frames required by the collective heap.

When a thread issues a request for heap storage, frames are attached to the initial thread. For RPC, this is the ECB that issued the `rpc_server_listen` API.

Note: These frames are not released until the initial ECB exits. For RPC, this means that the server has been shut down.

RPC C Header Files

Following is a list of standard RPC C header files that are supported by the TPF system:

```
c$thex.h
cobolbas.h
codestbh.h
dcemvs.h
dce/codesets.h
dce/cpconver.h
dce/csmgmt.h
dce/dce.h
dce/dceerror.h
dce/dcemsg.h
dce/dcemsgmsg.h
dce/dcerpcmsg.h
dce/dcerpcsvc.h
dce/dcesvc.h
dce/dcesvcmacro.h
dce/dcesvcmsg.h
dce/dlltypes.h
dce/idlbase.h
dce/idlddefs.h
dce/idles.h
dce/iovector.h
dce/lbase.h
dce/marshall.h
dce/nbase.h
dce/ncastat.h
dce/ndrold.h
dce/ndrrep.h
dce/ndrtypes.h
dce/rpcbase.h
dce/rpcexc.h
dce/rpcpvt.h
dce/rpctypes.h
dce/rpcxdl.h
dce/service.h
dce/stubbase.h
dce/twr.h
dce/uuid.h
ldrall.h
pthdex.h
```

```
rpcxstub.h
```

Note: Some of these RPC C header files are included by the server, the server stub file, or the server application.

Understanding Virtual Storage Access Method (VSAM) Database Support

This chapter describes the macro-level interface as well as the external interactions between the multiple virtual storage (MVS) system and the TPF system that are required to access VSAM databases from the TPF system. An introduction to VSAM is provided, including terminology that is useful in understanding *VSAM database support*, which is the term that we will use to refer to the support implemented in the TPF system that permits applications to access VSAM databases.

VSAM Concepts

VSAM is an IBM licensed program; as an access method service, it provides fast storage and retrieval of data. Records are stored in control intervals; that is, in the block on the disk that VSAM uses to store data records and control information that describes the records. Records are ordered by key values in a key field or by when they were stored.

For access to a keyed data set, also known as a key-sequenced data set (KSDS), you specify a key value; for access to a nonkeyed data set, also known as an entry-sequenced data set (ESDS) or sequential data set, you specify a relative byte address (RBA). A variation of keyed data set access is known as a relative record data set (RRDS), which is comprised entirely of fixed-length records; the record number is handled like a key.

VSAM maintains the concept of a logical next record and logical previous record interface to provide database scans and searches. Keyed positioning is also available to begin scans either at the beginning or in the middle of a data set, including a KSDS.

A KSDS has two data set components: a data component and an index component. The VSAM index of a highly volatile file may be very complex, requiring several levels of indexes. These layers are known as the *index set*. The index set points to the lowest level index, called the *sequence set*. For performance reasons, the sequence set typically resides on the same cylinder as the data control intervals to which it refers in the data component. The keys in the index set are always compressed. VSAM supports alternate indexes to provide alternate access to data; for example, the prime index key to a database could be an account number, while the alternate key could be a customer name.

The three ways to access data in a VSAM data set are:

- Direct access, which is usually required for keyed data sets
- Sequential access, which is usually required for entry-sequenced data sets
- Skip sequential access, which permits positioning to the next records based on a defined value.

Figure 34 on page 202 shows the typical layout of a key-sequenced data set.

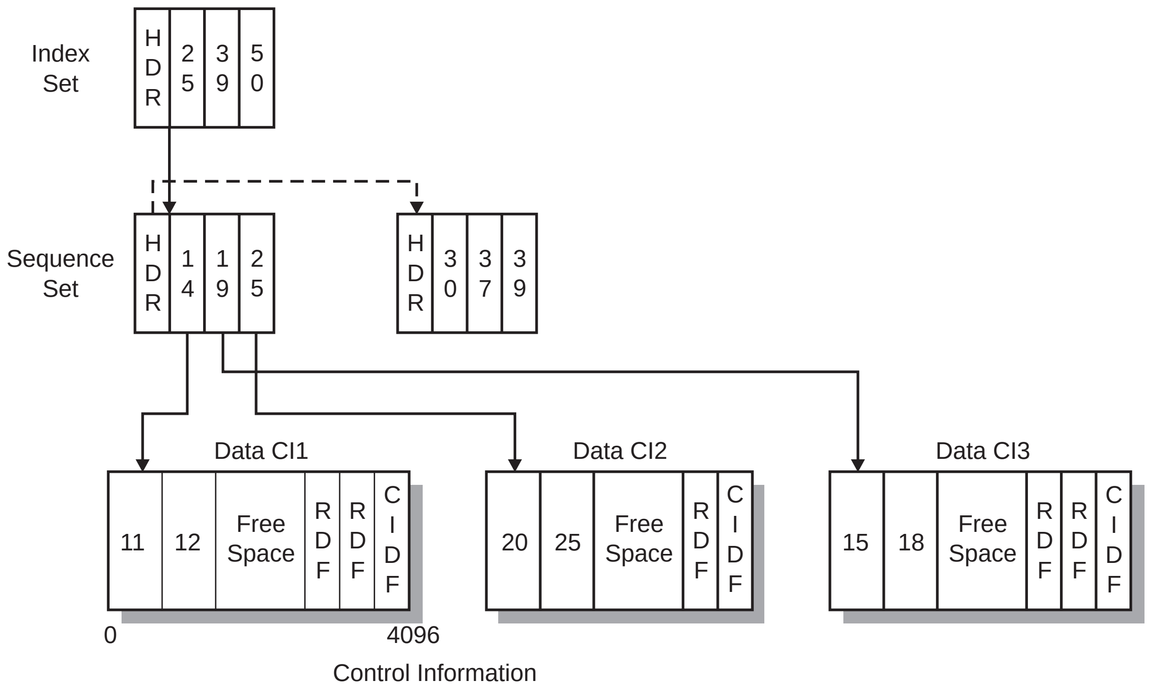


Figure 34. Layout of a Key-Sequenced Data Set

VSAM is controlled by a set of macros to manage data (OPEN, POINT, GET, PUT, ERASE, ENDREQ, and CLOSE) and a set of macros to manage control blocks (GENCB, TESTCB, SHOWCB, MODCB). These are described in MVS/XA VSAM Administration: Macro Instruction Reference.

Typically, an application requests VSAM services by running a predefined sequence of macro calls. The dialog is controlled by two major control blocks: the access method control block (ACB) and the request parameter list (RPL). These control blocks describe the characteristics of the data set, the type of access that is required, and the status of active requests. They are created using the TPF GENCB macro and are passed with each VSAM data request.

The dialog between VSAM and an application to read a record is structured as follows:

1. Enter the GENCB macro to create an ACB and an RPL.
2. Enter the OPEN macro to connect to a database.
3. Enter the GET macro to read a data record, followed by the CHECK macro to wait for results of the read.
4. Process the data record.
5. Enter the ENDREQ and CLOSE macros to end the request and disconnect from the database.

VSAM Database Support

The TPF system implements VSAM (called *VSAM database support*) through TPF general data set (GDS) support. This VSAM database support provides read-only access to VSAM KSDSs by using a macro interface and general processing protocol that is loosely modeled after IBM VSAM.

The VSAM database space is managed by an integrated catalog facility (ICF) entry that is set up on an MVS system at the time the VSAM data cluster is created. The catalog entry contains the physical characteristics of the cluster: the name of the data set, the extents, the number of volumes, the physical block size, the control interval size, and so on. In addition, the catalog entry contains the logical layout of the data and the type of organization. Because this information is required by the TPF system to decode VSAM requests, it is extracted from the MVS system by using the IDCAMS LISTCAT function (or other comparable function) and passed to the TPF system on the data set name. IDCAMS is an executable program in the MVS system that provides specialized VSAM support.

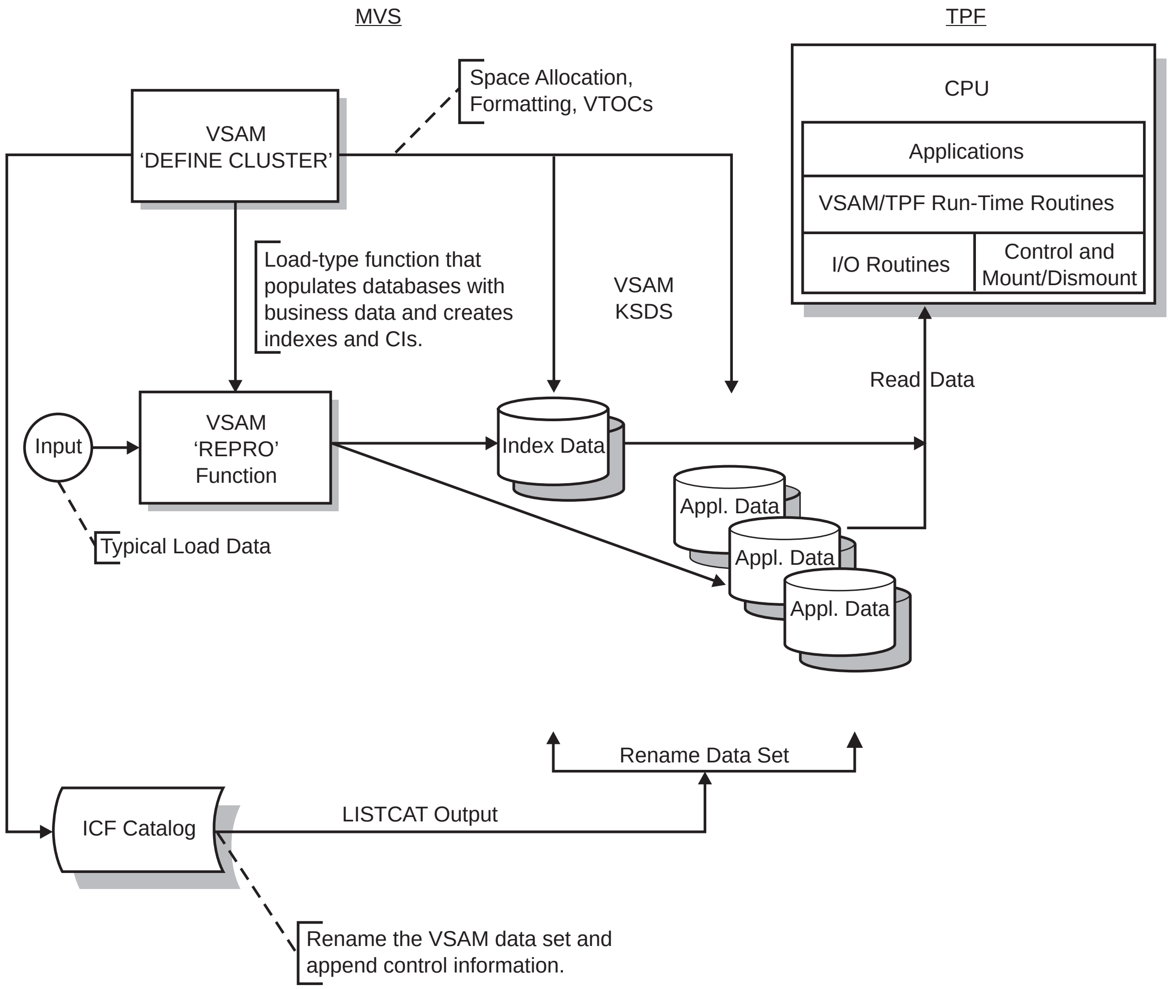
The actual management of space (that is, allocating data sets on DASD), multivolume control, populating disks with data, and performing index maintenance is performed by the MVS system. Each VSAM volume, therefore, is connected to the TPF system and MVS at the same time. Data is inserted into VSAM control intervals using the IDCAMS REPRO function on MVS; however, TPF applications cannot access a data set while it is being populated with data on the MVS system.

VSAM database support is grouped logically into four areas, as follows:

- Data set management (open, close, mount, and remove)
- Record management (servicing requests to manage data)
- Control block management (storage management and interface control)
- Input/output (I/O) management (drivers, index processing, decompression, and asynchronous control).

These areas are implemented in E-type programs called run-time routines that operate between application programs and the TPF system. Asynchronous control for I/O processing is not permitted; however, VSAM requests can be interleaved with application find and file requests to help database coexistence and migration.

Figure 35 on page 204 identifies the major components that comprise the interface and shows the logical flow between the components to convert a sample flat file into a VSAM key-sequenced data cluster and move it online to the TPF system.



MVS
TPF
Space Allocation, Formatting, VTOCs
VSAM 'DEFINE CLUSTER'
CPU
Applications
VSAM/TPF Run-Time Routines
I/O Routines
Control and Mount/Dismount
Load-type function that populates databases with business data and creates indexes and CIs.
VSAM KSDS
Read Data
Input
VSAM 'REPRO' Function
Index Data
Appl. Data
Appl. Data
Appl. Data
Typical Load Data
Rename Data Set
LISTCAT Output
ICF Catalog
Rename the VSAM data set and append control information.

Figure 35. VSAM Database Logical Flow

Disk Mirroring

VSAM database support provides mirrored disks as well as the ability to balance disk queues to provide a high level of data availability for TPF applications. With disk mirroring, a disk failure remains transparent to the application as long as the mirror disk can manage the request. For example, if a volume of an index or data space is taken offline either manually or by the TPF system, VSAM database support switches active read operations off the failed volume to the alternate volume on the mirror data cluster. In the same way, any read operations that occur after the disk failure are switched to the alternate disk. When the disk volume or cluster is brought online again, VSAM database support incorporates the disk or cluster into the configuration again and uses it immediately.

Disks can be added or removed from the configuration at any time by using the ZDSMG DM or MT command with the ALL parameter specified (for an entire cluster) or the VOLUME parameter specified (for an individual disk). Note that the index and data portions must be handled separately for a KSDS; therefore, you must enter the ZDSMG command twice, each time with a separate data definition (DD) name reference.

Each VSAM cluster must be identical to the other (except for the volume serial numbers (VOLSERs)) for disk mirroring to work. An MVS database administrator must run two jobs to create two VSAM clusters on different sets of disk volumes to set up a mirrored VSAM group. The MVS DEFINE command in the VSAM CREATE function is identical for both clusters except for the VOLSERs, which must change. This ensures that VSAM arranges the files in exactly the same way on both sets of disks. Each disk then has a unique MVS data set name.

Data Set Naming Convention

MVS catalog information is passed to the TPF system by appending it to the end of the data set name. Each MVS VSAM data set name of each portion of the VSAM cluster must end with the following:

```
.R <SS RBA> <High Level Block Number> .P <Key Position> L <Key Length> <.INDEX | .DATA>
```

SS RBA
Specifies the hexadecimal value of the sequence set RBA that is displayed in the LISTCAT output on the MVS system, divided by the control interval (CI) size. SS RBA is the pointer to the start of the sequence set CI blocks in the index set. For example, an SS RBA of 1 474 560 and a CI size of 4096 would result in X'168'.

High-Level Block Number
Specifies the 4-digit hexadecimal value of 1 plus the value of the high-level RBA that is displayed in a LISTCAT output on the MVS system, divided by the CI size. The high-level RBA is the pointer to the first index block in the highest-level index; that is, the point where all index lookups begin. For example, a high-level RBA of 8192 and a CI size of 4096 would have a block number of X'0003' 1+(8192/4096).

Key Position and Key Length
Specifies the hexadecimal offset to the key in the record, counting from 0. For example, a key of 22 bytes at byte offset 33 would result in .P21L16. These are the values specified in the KEYS parameter on the define cluster command; they are also the RKP and KEYLEN values that are displayed in LISTCAT on the MVS system.

INDEX | DATA
For a KSDS, VSAM (on MVS) automatically appends the names INDEX or DATA to the data set name during the VSAM create process to identify the index and data spaces of the cluster. For a cluster with a high-level RBA of 1 474 560, CI size of 4096, and key length of 22 bytes at offset 33 in each record, the data set name for the index portion of the cluster would be `.R1680003.P21L16.INDEX`. The data portion of the cluster would be `.R1680003.P21L16.DATA`.

Referring to Data Definition (DD) Names and Clusters

The TPF system provides several levels of references for data sets to maintain a high level of flexibility in managing files. By convention, applications refer to the VSAM database by using an application DD name, while the TPF system refers to the VSAM database by using a system DD name, which is derived for the application DD name. The TPF system can also refer to a disk by its cluster data set name and volume sequence number. See “Constructing the VSAM Cluster Data Set Names” on page 214 for more information about constructing cluster data set names.

Each key-sequenced data set cluster is then defined by two unique DD names: one to identify the index space and one to identify the data space. Therefore, there are two unique data set names. In a mirrored cluster, there are four unique DD names and four unique data set names. A cluster group refers to the entire group of VSAM clusters that comprise a prime and mirror set.

DD names are created by first assigning an 8-byte name to an application to identify the cluster group (prime and mirror) (for example, FREQFLYR). The application uses this name in the ACB to refer to the cluster group. The TPF system refers to this DD name (using the ZDSMG command) by appending INPR, INMI, DAPR, or DAMI when referring to discrete parts of the cluster group. Consider the following example.

FREQFLYRINPR	Specifies the prime index portion of the cluster group.
FREQFLYRDAPR	Specifies the prime data portion of the cluster group.
FREQFLYRINMI	Specifies the mirror index, a copy of INPR.
FREQFLYRDAMI	Specifies the mirror data, a copy of DAPR.

The disk volumes in a DD name can also be addressed by the ZDMSG command with the VOL parameter specified.

Table 18 shows how a cluster referencing scheme might look. Application programmers use the application DD name when writing programs. A TPF operator uses the TPF system DD name to mount and remove files, while an MVS database administrator uses the MVS data set names to create and copy VSAM data sets.

Table 18. VSAM Database Support Cluster Reference

Application DD Name	TPF System DD Name	MVS Data Set Name
CHECKING	CHECKINGINPR	TPF.CHECKING.PRIME.R1680003.P21L16.INDEX
CHECKING	CHECKINGDAPR	TPF.CHECKING.PRIME.R1680003.P21L16.DATA
CHECKING	CHECKINGINMI	TPF.CHECKING.MIRROR.R1680003.P21L16.INDEX
CHECKING	CHECKINGDAMI	TPF.CHECKING.MIRROR.R1680003.P21L16.DATA

I/O Interface and VFA

The I/O management scheme used by the interface uses several optimization options that include virtual file access (VFA) buffering for index blocks (CIs). All index blocks in the index data set are VFA candidates. Each time an index block is retrieved, the run-time program tries to get it from VFA. If it is not present, it is retrieved from disk and placed in VFA for later retrieval. The number of index blocks in the IMBED type of data cluster is typically small.

The run-time program uses the standard TPF macro interface to read data blocks and perform I/O. The GDSNC and FINWC macros are used to read index set, sequence set, and data set control intervals. Each call converts the index set vertical pointer to a relative record number that the TPF system can convert into a file address reference format (FARF) type of address.

For a typical direct key retrieval for an IMBED type of data cluster, this requires the TPF system to perform two VFA reads of the index set, followed by two disk I/Os: one to read the sequence set CI and one to read the data record CI. No head movement is required between the two physical I/Os because the sequence set CI is at the same seek cylinder address as the data record CI.

Scanning, or sequential processing, is highly optimized because the run-time program takes full advantage of the sequence set to data record relationship, including scanning within a CI that was already retrieved and then scanning the entire cylinder to read all the records in a CI. To take full advantage of this, an application may spawn many entry control blocks (ECBs) to consume an entire VSAM file in parallel.

VSAM Database Constraints

VSAM database support for the TPF system is read-only. This means that while a VSAM data set is being actively referenced by TPF applications, the MVS system will not try to modify the data set, and the TPF system will not write to or modify the VSAM data set in any way. The MVS system, however, can modify the database when the data cluster is logically or physically disconnected from the TPF application that is referring to it. Access to VSAM data sets between the MVS and TPF systems, therefore, is mutually exclusive.

The following VSAM database macros support this read-only interface:

Table 19. VSAM Database Support Macros

VSAM Database Macro	Description	MVS System Equivalent Macro
VOPNC	Use this macro to connect to a VSAM database.	OPEN
VGETC	Use this macro to read a record.	GET
VCHKC	Use this macro to wait for a request to end.	CHECK
VPNTC	Use this macro to position for access.	POINT
VENDC	Use this macro to end a request.	ENDREQ
VCLSC	Use this macro to disconnect from a database.	CLOSE
VGENC	Use this macro to generate a control block.	GENCB
VSHOC	Use this macro to access a control block.	SHOWCB

Because VSAM database support for the TPF system is read-only, the MACRF and OPTCD parameters of the VGENC macro, which specify processing options and type of access, are restricted to the read-only set and exclude parameters that would change the database. The locate (LOC) and move (MVE) parameters are supported by VSAM database support for retrieving data. The VGENC macro is also used to generate ACBs and RPLs. The maximum number of ACBs that can be specified per ECB is 16; the maximum number of RPLs per ECB that can be specified is 32.

Three types of access to a VSAM data set were previously mentioned in describing VSAM:

- Direct access is supported for transaction-type requests.
- Sequential access is supported for scan-type requests.
- Skip sequential access to a VSAM data set is ***not*** supported for the TPF system.

The ICF catalog structure on the MVS system is the only catalog structure that is supported. The CI must be set to the maximum TPF block size of 4096 bytes and the physical block size on a track must be equal to the CI size. Alternate indexes and the specialized linear data set are not supported.

The prime and backup copies of each data cluster must be physical and logical mirror images of each other for applications that require backup. Each disk must be a physical mirror image of its alternate.

You must specify either the IMBED or the NOIMBED option for the DEFINE command when creating the VSAM database on the MVS system. The maximum size of a CI is 4096 bytes; spanning of CIs is not supported. The maximum record size, therefore, is 4089 bytes after taking into account the 4-byte control interval definition field (CIDF) at the end of the block and a 3-byte record definition field (RDF) field that follows it. The CIDF describes the free space, if there is any, in the control interval. The RDF describes the characteristics of each record.

The index component cannot span multiple volumes and can only be on one extent.

The following options are restricted for the DEFINE CLUSTER command on the MVS system when you are defining VSAM data sets that will be used by the TPF system:

- NONSPANNED
- NOREPLICATE
- CISZ
- CYLINDERS
- UNIQUE
- INDEXED.

VSAM Database Considerations

VSAM database support is compatible with certain levels of data facility product (DFP) VSAM, MVS/ESA, and TPF; see *TPF Migration Guide: Program Update Tapes* for more information about what levels are supported. VSAM database support coexists with all other TPF database support.

The VSHOC macro is a special version of the VSAM SHOWCB macro; this macro returns the first ACB and RPL in sequence. Subsequent control blocks of the same type are accessed by using the chain word in each block.

VSAM blocks do not use the format flag; therefore, you must review code that depends on format flags. The area that was occupied by the format flag is used to hold part of the CDF field at the end of the block.

Storage allocation for control blocks is managed by TPF VSAM database database support or by the application (by coding the WORKAREA parameter on the GENCB macro). Storage allocation for I/O buffers is managed exclusively by VSAM database support by using the GETCC macro. As a result, the buffer parameters that are usually coded on the GENCB macro in the MVS system are not required in the TPF system. Instead, a new parameter, LEVEL=D*x*, is coded in generating the RPL to specify the TPF data level and core block reference word (CBRW) to use for I/O operations.

For performance reasons, all I/O requests in VSAM database support are synchronous, which means that the OPTCD=ASY parameter is not supported for the VGENC macro. Therefore, all VGETC, VENDC, and VPNTC requests do not need the corresponding VCHKC macro to get the results of the macro operation.

VSAM database support does not include the EXLST macro to generate an EXIT list.

VSAM Database Support with Other Utilities

VSAM data sets are mounted to the TPF system as 4-KB general data sets (GDSs). These data sets are managed by the TPF control program as GDSs and are mounted and removed using GDS commands and macros. By using the GDS interface, VSAM data sets inherit all of the reliability, availability, and serviceability characteristics that are provided by the GDS packaging, including I/O error recovery, environmental error record editing and printing program (EREP) analysis, and sustained connections across TPF system restarts. VSAM database support additionally enhances data availability through the use of disk mirroring, as mentioned previously.

TPF data collection tools, which measure I/O performance, storage use, and response times, are available with VSAM database support to use in capacity planning and performance measurement.

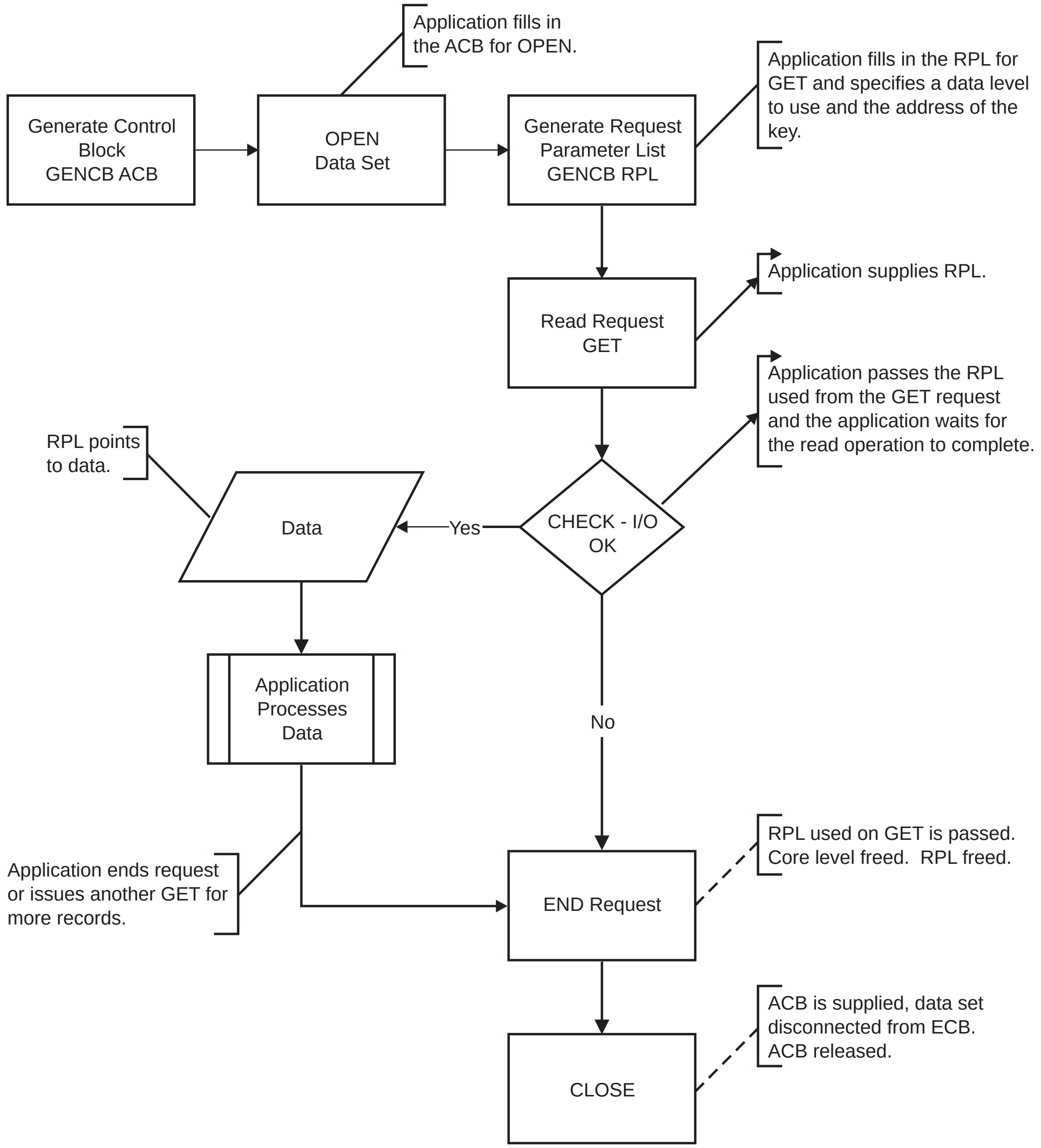
To extract the required ICF catalog information for the TPF system, a COBOL utility is available, on request, that demonstrates how to derive the catalog information that is needed by the TPF system as part of the data set name. A sample restructured extended executor language (REXX) program is provided in “Constructing the VSAM Cluster Data Set Names” on page 214, which can be used to construct the VSAM database support cluster data set names to be mounted to the TPF system.

VSAM Database Support Request Flow Control

An ECB communicates with VSAM database support through a set of macros to manage a VSAM database. The flow of control begins by running the VGENC macro to create an access method control block (ACB). Now that the data cluster to be opened is defined, as well as the processing options that are required, the ECB connects to the VSAM database by using the VOPNC macro. This prepares the data set for access. For VOPNC requests, the run-time routines internally issue the TPF GDSNC macro to connect the ECB to the database and save the output from that process.

Note: The application DD name is used to derive the TPF system DD name that is used when the cluster data set components are mounted to the TPF system. In this way, VSAM database support is able to make the connection to the prime and mirror cluster data sets.

Using an RPL, the ECB then manages records in the database by using the VGETC and VPNTC macros. Use the VCHKC macro to get the results of requests and the VENDC macro to end requests. When ECB processing is completed, it issues a VCLSC macro to break the connection. This flow of control is similar to a typical TPF database access where an ECB hashes a key to a fixed record slot, calls the FACS program to get the disk address of the slot followed by a find and wait to get the data block, and finally issues the RELCC macro to release it. The difference between that flow control and VSAM database support is that VSAM database support provides functions such as indexing and hashing in addition to the ability to locate specific records in blocks rather than having to write code for it. VSAM database support also performs complex, hierarchical data searches and full database scans. Figure 36 on page 210 shows the interaction of VSAM macros and control blocks and how they are used during request processing.



Application fills in the ACB for OPEN.
Application fills in the RPL for GET and specifies a data level to use and the address of the key.
Generate Control Block GENCB ACB
OPEN Data Set
Generate Request Parameter List GENCB RPL
Application supplies RPL.
Read Request GET
Application passes the RPL used from the GET request and the application waits for the read operation to complete.
RPL points to data.
Data
Yes
CHECK - I/O OK
Application Processes Data
No
RPL used on GET is passed. Core level freed. RPL freed.
Application ends request or issues another GET for more records.
END Request
ACB is supplied, data set disconnected from ECB. ACB released.
CLOSE

Figure 36. VSAM Database Support Request Flow Example

As you see in Figure 36, data requests to VSAM database support are asynchronous and permit the interleaving of native I/O requests with VSAM database support requests. A VSAM database I/O request comprises the TPF GDSNC macro and a FINWC macro call. The RBA for input to the GDSNC macro is obtained inside run-time processing by searching the index set and the sequence set. The GDSNC macro converts the RBA to a TPF disk address and then issues the FINWC macro to read the data to the ECB data level (D*x*) that was specified in the RPL.

Applications issue the VCHKC macro to get the results of an I/O request; the completion status of the request is placed in register 15. Detailed error information is placed in the feedback field of the RPL. The VCHKC macro causes the following to occur:

- An internal TPF WAITC macro is issued.
- The RPL is filled.
- Register 15 is set.

Because all I/O is synchronous, the VCHKC macro is not required to interrogate the status of the VGETC request. The same output is provided on a VGETC request as for a VCHKC request. The VCHKC macro is available for compatibility.

VSAM database support has no requirement for record locking. However, because a VSAM VGETC macro request results in a TPF FINWC macro request, the existing TPF page-level record hold feature may easily be added if locking is required at any point. Currently, all data records are shared by all ECBs. VSAM data sets are attached to the TPF system by using the ZDSMG command and the standard TPF mount data set procedure in the GDS.

Sample VSAM Database Support Program

Following is a sample program to show the VSAM database support macros.

```
     BEGIN NAME=xxxx,VERSION=41
     VSACB ,                                 ACB DSECT definition
     VSRPL ,                                 RPL DSECT definition
     VGENC        BLK=ACB,AM=VSAM,DDNAME=CUSTOMER,
                  MACRF=(KEY,DIR,IN)
     ST           R1,EBX000                  Save ACB address

     VGENC        BLK=RPL,AM=VSAM,ACB=EBX000,LEVEL=D2,AREA=EBX008,
                  ARG=EBW000, OPTCD=(LOC,SYN)
     IF CC0
     THEN,
            L     R1,EBX000                  Pick up ACB address
            VOPNC (R1)
            IF CC0                           If VOPNC ok
            THEN,

                  MVC       EBW000,=C'00'       Set up Arg fields.
                  MVC       EBW002,=C'12345678' Account Number.
                  VGETC     RPL=EBX008
                  IF CC0,

                  ... A sample TPF native I/O call is intermingled here.

                  MVC       CE1FA3,=C'PR'       Record ID.
                  MVC       CE1FM3(4),FARF3 Address
                  FINWC     D3,ERRLAB           Find a TPF record.

                  ... Now back to the VSAM call
                  VCHKC     RPL=EBX008
                  IF CC0,
                  THEN,
                        CE1CR2 now contains the VSAM record.
                        The data record in CE1CR2 is pointed to by EBX008.
                        CE1CR3 contains the TPF record.
                  ENDIF,

ERRLAB            DS    0H
                  VENDC RPL=EBX008
                  L     R1,EBX000
                  VCLSC EBX000

            ENDIF ,
     ENDIF ,
```

Managing Buffers

The application interface to VSAM database support is pointer-driven using the ACB and RPL control structures. The ACB is used as context to control the interaction between the ECB and VSAM database support. It is created by the VGENC macro and is used for all interactions. The ACB is released through the VCLSC macro. The RPL is used to control individual VSAM database data requests. It is also used to hold parameters and to provide the status of requests and responses. The VENDC macro is used to release the RPL.

For a typical VGETC macro request, the application sets pointers in the RPL to the type of request to be performed in addition to any special processing options. The result of the VGETC macro is returned in the RPL by the run-time routines. Get and release macros (GETCC and RELCC), with the DETAC macro, acquire and release storage for ACBs and RPLs. For multiple, concurrent requests in a single ECB, the application specifies one unique RPL for each request. This is done by coding one VGENC macro for each request. VSAM database support also uses TPF working storage for output data buffers. Additionally, VFA is used to store and retrieve index CIs.

When an application calls its first VGENC macro to generate a VSAM control block, VSAM database support gets a 4-KB block and detaches it from the data level (DF). The block is used to hold a context data block and subsequent interface control blocks, ACBs, and RPLs. Space is optimized in the buffer by stacking blocks from the bottom up. Interface blocks are allocated in the buffer through VGENC macro calls; they are unallocated by using VENDC macro calls. The 4-KB block is released only when the ECB exits. This optimizes block management because subsequent calls of VSAM by the same ECB do not need to allocate the context block.

Sequence set CIs are the lowest-level index blocks; they reside on the same cylinder as the data records for performance reasons. These CI blocks are retrieved and detached on the data level specified in the RPL. They are kept for subsequent fast scan requests and then are released by using the VENDC macro. For a single direct key retrieval, the data level contains a private copy of the sequence set block (detached) and a private copy of the VSAM data record block (also detached). Applications that need to economize on storage would use the LOCate mode retrieval method to get access to a data record rather than MOVE mode, which moves the data record out of the VSAM data buffer into an application area. The VSAM data record block remains detached from that data level until a VENDC macro is issued for the RPL that owns the request. The run-time interface requires 12 KB of space for a typical direct key retrieval.

Reusing Data Levels

Applications can reuse the data level in the ECB that is used to perform VSAM I/O operations (as indicated in the RPL) after a VENDC macro is issued for the VSAM request that was using the data level. This avoids the necessity to repeatedly call the VOPNC macro in centralized application programming interface (API) routines. The rule for reusing a data level and core block is that the level is freed by the application before calling the first VGETC macro and that a VENDC macro request must be called before an application can reuse the level. The VENDC macro releases any positioning and context for the current VSAM database request.

Return Codes

Register 15 (R15) contains the return code for VSAM database support macro processing; the condition code is set on return from each macro call. The interface for VSAM database support return codes is the same as for the MVS system, as follows:

- For data set open and close errors, the ACB ERROR field contains the detailed reason for the error.
- For record processing (for example, the VGETC and VPNTC macros), the RPL FEEDBACK field contains the detailed description of the error.
- For the VGENC macro, R14 contains the reason code.

For performance reasons, VSAM database support provides these codes immediately following macro processing; you do not need to enter the VSHOC macro call. (This is different from the MVS system, where you would need to enter the SHOWCB macro call.) When a record is not found, the return code is set to **not found** instead of showing successful or null status. VSAM database does not support options for error message text and codes.

Table 20. VSAM Database Support Return Codes

Return Code	Meaning
0	Macro processing was successful.
1	Request was not valid.
2	Block type was not valid.
3	Parameter was not valid.
8	VSAM database support internal work area is full.
9	VSAM database support internal work area is too small.
14	Combination of options is not valid.
15	Address not on a fullword boundary.
16	End of file.
17	Record not found.
18	Duplicate previous operation.
32	VSAM database support data set name is not valid.
33	VSAM database support data set open error.
34	VSAM database support data set error during read.
35	VSAM database support data set FINWC macro I/O error.
44	Short length record on MVE.
128	Mask for retry-type errors.
129	VSAM database support index error.

Error Recovery

Because VSAM database support consists primarily of E-type programs that run between application programs and the TPF control program (CP), all error recovery functions that are provided by the CP are available to applications using VSAM database support. These include DASD single-image I/O error handling, recovery from an abnormal ending of a program, exit processing, and system recovery.

Constructing the VSAM Cluster Data Set Names

The following is a sample REXX program that you can use to construct the VSAM database support cluster data set names to be mounted to the TPF system. The TPF system requires that a subset of the catalog information be passed to it in the data set name to be able to navigate the VSAM cluster in satisfying requests for data records made by TPF VSAM applications.

```
/* REXX */
/********************************************************************/
/*                                                                  */
/*       NAME: VSAMTPFX (SAMPLE CSI PROGRAM BASED ON IGGCSIRX)      */
/*                                                                  */
/*DESCRIPTION: THIS REXX EXEC CAN BE USED TO CALL THE CATALOG       */
/*             SEARCH INTERFACE AND GENERATE THE NECESSARY          */
/*             DATA SET NAME SUFFIX THAT CAN BE APPENDED TO THE     */
/*             CLUSTER/COMPONENTS IN ORDER TO COMMUNICATE CATALOG   */
/*             INFORMATION TO THE TPF/VSAM SUPPORT WHEN MOUNTING    */
/*             THE CLUSTER TO TPF.                                  */
/*                                                                  */
/*      INPUT: FILTER KEY (KSDS CLUSTER NAME)                       */
/*             IN THE FORM *.DDNAME.*                               */
/*                 WHERE DDNAME IS 8 CHARACTERS IN LENGTH AND       */
/*                 CAN REPRESENT THE FIRST 8 CHARACTERS OF THE 12   */
/*                 CHARACTER SYSTEM DDNAMES USED BY TPF TO REFERENCE*/
/*                 THE DATASET COMPONENTS OF THE PRIME/MIRROR       */
/*                 CLUSTERS MOUNTED TO TPF.                         */
/*                                                                  */
/*     OUTPUT: ALTER DATA SET NAME INFORMATION                      */
/*             CONSISTS OF THE NEW DATA SET NAME APPENDED WITH      */
/*             ENCODED CATALOG INFORMATION DERIVED FROM THE INPUT   */
/*             DATA SET NAME CLUSTER CATALOG ENTRY.                 */
/*                                                                  */
/* EXAMPLE:                                                         */
/*                                                                  */
/*    CLUSTER NAME     => *.CHECKING                                */
/*    INDEX COMPONENT  => *.CHECKING.INDEX                          */
/*    DATA  COMPONENT  => *.CHECKING.DATA                           */
/*                                                                  */
/*                              (values in hexadecimal)             */
/*                                                                  */
/*    CATALOG LIST ENCODING =>  R 168 0001 . P22 L16                */
/*                              |   |    |     |   |                */
/*                       marker |   |    |     |   | Key length     */
/*                                  |    |     | Key Position       */
/*                       SS RBA/4096|    |                          */
/*                                       | 1+(HIGH LEVEL RBA/4096)  */
/*                                                                  */
/*                                                                  */
/*                                                                  */
/*    NEW CLUSTER NAME => *.CHECKING.R1680001.P22L16                */
/*    INDEX COMPONENT  => *.CHECKING.R1680001.P22L16.INDEX          */
/*    DATA  COMPONENT  => *.CHECKING.R1680001.P22L16.DATA           */
/*                                                                  */
/* TPF MOUNT INFORMATION:                                           */
/*                                                                  */
/* ZDSMG MT {SDA} CHECKINGINPR DSN-*.CHECKING.R1680001.P22L16.INDEX */
/* ZDSMG MT {SDA} CHECKINGDAPR DSN-*.CHECKING.R1680001.P22L16.DATA  */
/*                                                                  */
/*                                                                  */
/********************************************************************/
  SAY 'ENTER VSAM CLUSTER NAME'      /* ASK FOR FILTER KEY          */
  PULL KEY                           /* GET FILTER KEY              */
/********************************************************************/
/*                                                                  */
/*  INITIALIZE THE PARM LIST                                        */
/*                                                                  */
```

```
 /********************************************************************/
MODRSNRC = SUBSTR(' ',1,4)          /*   CLEAR MODULE/RETURN/REASON  */
CSIFILTK = SUBSTR(KEY,1,44)         /*   MOVE FILTER KEY INTO LIST   */
CSICATNM = SUBSTR(' ',1,44)         /*   CLEAR CATALOG NAME          */
CSIRESNM = SUBSTR(' ',1,44)         /*   CLEAR RESUME NAME           */
CSIDTYPS = SUBSTR(' ',1,16)         /*   CLEAR ENTRY TYPES           */
CSICLDI  = SUBSTR('Y',1,1)          /*   INDICATE DATA AND INDEX     */
CSIRESUM = SUBSTR(' ',1,1)          /*   CLEAR RESUME FLAG           */
CSIS1CAT = SUBSTR(' ',1,1)          /*   INDICATE SEARCH > 1 CATALOGS*/
CSIRESRV = SUBSTR(' ',1,1)          /*   CLEAR RESERVE CHARACTER     */
CSINUMEN = '0004'X                  /*   INIT NUMBER OF FIELDS       */
CSIFLD1    = SUBSTR('AMDKEY',1,8)              /* AMDKEY             */
CSIFLD1    = CSIFLD1 || SUBSTR('AMDCIREC',1,8) /* AMDCIREC           */
CSIFLD1    = CSIFLD1 || SUBSTR('HKRBA',1,8)    /* HKRBA              */
CSIFLD1    = CSIFLD1 || SUBSTR('HARBA',1,8)    /* HARBA              */
 /********************************************************************/
 /*                                                                  */
 /*  BUILD THE SELECTION CRITERIA FIELDS PART OF PARAMETER LIST      */
 /*                                                                  */
 /********************************************************************/
CSIOPTS  = CSICLDI || CSIRESUM || CSIS1CAT || CSIRESRV
CSIFIELD = CSIFILTK || CSICATNM || CSIRESNM || CSIDTYPS || CSIOPTS
CSIFIELD = CSIFIELD || CSINUMEN || CSIFLD1

 /********************************************************************/
 /*                                                                  */
 /*  INITIALIZE AND BUILD WORK AREA OUTPUT PART OF PARAMETER LIST    */
 /*                                                                  */
 /********************************************************************/
WORKLEN = 1024
DWORK = '00000400'X || COPIES('00'X,WORKLEN-4)

 /********************************************************************/
 /*                                                                  */
 /*  INITIALIZE WORK VARIABLES                                       */
 /*                                                                  */
 /********************************************************************/
RESUME = 'Y'
CATNAMET = SUBSTR(' ',1,44)
DNAMET = SUBSTR(' ',1,44)

 /********************************************************************/
 /*                                                                  */
 /*  SET UP LOOP FOR RESUME (IF A RESUME IS NECESSARY)               */
 /*                                                                  */
 /********************************************************************/
DO WHILE RESUME = 'Y'

 /********************************************************************/
 /*                                                                  */
 /*  ISSUE LINK TO CATALOG GENERIC FILTER INTERFACE                  */
 /*                                                                  */
 /********************************************************************/
 ADDRESS LINKPGM 'IGGCSI00  MODRSNRC  CSIFIELD  DWORK'

 RESUME = SUBSTR(CSIFIELD,150,1)    /* GET RESUME FLAG FOR NEXT LOOP */
 USEDLEN = C2D(SUBSTR(DWORK,9,4))   /* GET AMOUNT OF WORK AREA USED  */
 POS1=15                            /* STARTING POSITION             */

 /********************************************************************/
 /*                                                                  */
 /*  PROCESS DATA RETURNED IN WORK AREA                              */
 /*                                                                  */
 /********************************************************************/
 DO WHILE POS1 < USEDLEN            /* DO UNTIL ALL DATA IS PROCESSED*/
   IF SUBSTR(DWORK,POS1+1,1) = '0'  /* IF CATALOG, PRINT CATALOG HEAD*/
    THEN DO
```

```
        CATNAME=SUBSTR(DWORK,POS1+2,44)
        IF CATNAME ^= CATNAMET THEN /* IF RESUME NAME MAY ALREADY BE*/
         DO                         /*    PRINTED                   */
          SAY 'CATALOG ' CATNAME    /* IF NOT, PRINT IT             */
          SAY ' '
          CATNAMET = CATNAME
         END
        POS1 = POS1 + 50
        END

  DNAME = SUBSTR(DWORK,POS1+2,44)  /* GET ENTRY NAME                */

/********************************************************************/
/*                                                                  */
/*  ASSIGN ENTRY TYPE NAME                                          */
/*                                                                  */
/********************************************************************/
  IF SUBSTR(DWORK,POS1+1,1) = 'C' THEN DTYPE = 'CLUSTER '
   ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'D' THEN DTYPE = 'DATA    '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'I' THEN DTYPE = 'INDEX   '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'A' THEN DTYPE = 'NONVSAM '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'H' THEN DTYPE = 'GDS     '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'B' THEN DTYPE = 'GDG     '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'R' THEN DTYPE = 'PATH    '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'G' THEN DTYPE = 'AIX     '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'X' THEN DTYPE = 'ALIAS   '
    ELSE
     IF SUBSTR(DWORK,POS1+1,1) = 'U' THEN DTYPE = 'UCAT    '
    ELSE
     DTYPE = '        '
/********************************************************************/
/*                                                                  */
/*  HAVE NAME AND TYPE, IF INDEX ENTRY CALCULATE MAGIC PARAMETERS   */
/*                                                                  */
/********************************************************************/
   POS1 = POS1 + 46
   IF DTYPE = 'INDEX'
    THEN DO

      POSLEN1 = POS1    + 4
      POSLEN2 = POSLEN1 + 2
      POSLEN3 = POSLEN2 + 2
      POSLEN4 = POSLEN3 + 2

      POSDAT1 = POSLEN4 + 2
      POSDAT2 = POSDAT1 + C2D(SUBSTR(DWORK,POSLEN1,2))
      POSDAT3 = POSDAT2 + C2D(SUBSTR(DWORK,POSLEN2,2))
      POSDAT4 = POSDAT3 + C2D(SUBSTR(DWORK,POSLEN3,2))

      SAY "KEY-POS      => X'"C2X(SUBSTR(DWORK,POSDAT1,2))"'"
      KEYPOS            =     C2X(SUBSTR(DWORK,POSDAT1,2))

      SAY "KEY-LEN      => X'"C2X(SUBSTR(DWORK,POSDAT1+2,2))"'"
      KEYLEN =                C2X(SUBSTR(DWORK,POSDAT1+2,2))
      SAY "CI-SIZE      => X'"C2X(SUBSTR(DWORK,POSDAT2,4))"'"
      CISIZE            =     C2D(SUBSTR(DWORK,POSDAT2,4))

      SAY "HI-LEVEL-RBA => X'"C2X(SUBSTR(DWORK,POSDAT3,4))"'"
      HLRBA             =     C2D(SUBSTR(DWORK,POSDAT3,4))
      SAY "SEQ-SET-RBA  => X'"C2X(SUBSTR(DWORK,POSDAT4,4))"'"
      SSRBA             =     C2D(SUBSTR(DWORK,POSDAT4,4))

      MAGIC =          'R' || D2X((SSRBA/CISIZE),3)
      MAGIC = MAGIC || D2X(((HLRBA/CISIZE)+1),4) || '.'
      MAGIC = MAGIC || 'P' || RIGHT(KEYPOS,2)
      MAGIC = MAGIC || 'L' || RIGHT(KEYLEN,2)

      SAY ''
      SAY 'INPUT CLUSTER NAME         <= ' KEY

      SAY ''
      SAY "CATALOG SUFFIX STRING      == '"MAGIC"'"

      ALTERDSN = TRANSLATE(KEY,' ','.')
      IF WORDS(ALTERDSN) >= '2'
       THEN DO

        NEWDSN = SUBWORD(ALTERDSN,1,1) || '.'

        DDNAME = SUBWORD(ALTERDSN,2,1)
        IF LENGTH(DDNAME) <8
        THEN DDNAME = DDNAME || '*'

        NEWDSN = NEWDSN || DDNAME || '.'
        NEWDSN = NEWDSN || MAGIC

        SAY ''
        SAY "NEW CLUSTER NAME           => '"NEWDSN"'"
        SAY "NEW DATA  COMPONENT NAME   => '"NEWDSN || .DATA"'"
        SAY "NEW INDEX COMPONENT NAME   => '"NEWDSN || .INDEX"'"
        SAY ''

        DAPRDDN = DDNAME || ' || DAPR'
        INPRDDN = DDNAME || ' || INPR'
        DAMIDDN = DDNAME || ' || DAMI'
        INMIDDN = DDNAME || ' || INMI'

        SAY 'TPF (SYSTEM) DDNAMES:'
        SAY ''
        SAY "  DATA  COMPONENT (PRIME)  => '"DAPRDDN"'"
        SAY "  INDEX COMPONENT (PRIME)  => '"INPRDDN"'"
        SAY ''
        SAY "  DATA  COMPONENT (MIRROR) => '"DAMIDDN"'"
        SAY "  INDEX COMPONENT (MIRROR) => '"INMIDDN"'"
        SAY ''

        SAY 'TPF (APPLICATION) DDNAME:'
        SAY ''
        SAY "  '"DDNAME"'"

        IF POS("*",DDNAME) \= 0
        THEN DO
          SAY ''
          SAY '* = MUST BE 8 CHARACTERS IN LENGTH FOR VSAM/TPF USE'
        END
    END
  END

/*********************************************************************/
/*                                                                   */
/*   GET POSITION OF NEXT ENTRY                                      */
/*                                                                   */
/*********************************************************************/
```

```
    POS1 = POS1 + C2D(SUBSTR(DWORK,POS1,2))
  END
END
/* END  OF PROGRAM */
```

Coding Your Own Library Functions

In addition to the library functions provided with the IBM C and C++ compiler products on the System/390 platform and those added for the TPF system, you can code your own library functions, either in C language or in assembler. There are different conventions for each.

Coding Library Functions in C

There are no special requirements for coding library functions in C. The procedures for compiling and loading library functions to the TPF system is the same as compiling and loading application programs written in C. See “Customizing C/C++ Language Support” on page 301 for more details.

Coding Library Functions in Assembler

There are special requirements for coding library functions in assembler. The first of these is knowing which registers are available and which are reserved for the TPF system. The second is understanding the prolog and epilog macros.

There are two kinds of assembler programs: those that call additional TPF services and those that do not. Programs that call additional services require prologs and epilogs. Programs that do not call additional services do not require prologs or epilogs.

For functions requiring prologs and epilogs, when the assembler program begins to run, the first statement is a BEGIN statement, coded with TPFISOC=YES. A frame is not automatically added to the call stack when an assembler program is run but if the assembler program calls other programs, the environment must be saved first, to retain the information needed to return to the original C function. The prolog macro, TMSPC, saves the C environment. The application base needed for assembler applications, CE1SVP, must be loaded before calling the entry point. Before a return statement (BACKC or EXITC macros), the epilog, a TMSEC macro, must be coded to restore the C environment.

Register Conventions

TPF has its own set of register conventions, which have been adapted for use by the IBM C compiler. Some registers are reserved for TPF system use while others are used by the compiler for specific purposes. The register conventions for ISO-C and TARGET(TPF) are somewhat different. Most registers are available for application use and are saved and restored across function calls. Some registers are used by the compiler. These are not saved or restored across calls.

The following are summaries. See the *TPF Program Development Support Reference* for more information about register conventions.

Registers	ISO-C Convention
R0	Available for application program use
R1	Available for application program use Used by the compiler for parameter list pointer.
R2	Available for application program use.
R3	Available for application program use. Used by the compiler as the code base.

Registers	ISO-C Convention
R4 – R11	Available for application program use.
R12	Address of the TCA.
R13	Address of the DSA.
R14	Available for application program use. Used by the compiler as the link register.
R15	Available for application program use. Used by the compiler as the called function address and as the return register.

Register	TARGET(TPF) Convention
R0 – R5	Available for application program use.
R6	Parameter list pointer/function return value.
R7	Reserved, pointer to current stack frame.
R8	Reserved, program base.
R9	Reserved, ECB base register
R10	Available for application use but not saved across function calls.
R11	Reserved, always contains X'1000'
R12	Available for application use but contains X'2000' across external function calls and returns.
R13	Available for application use but contains TPF system stack pointer across function calls.
R14	Available for application program use, on function calls is used as the return address register.
R15	Available for application program use, used in external and library function linkage.

C Language Support Prologs

Library functions written in assembler, as well as external functions linked to DLMs written in assembler, that need to acquire a stack frame, must begin with a TMSPC macro call when called from ISO-C functions or with an ICPLOG macro call when called from TARGET(TPF) functions. The TMSPC or ICPLOG macros must be the first instruction coded after the BEGIN macro. The code generated by these macros helps manage the programming environment of the C caller.

Earlier we talked about the need for stack blocks and stack frames. C functions use a stack for storage of local variables and parameter lists. The first time a C function is called, a stack block is attached to the ECB. In addition to saving registers, the TMSPC and ICPLOG macros are used to allocate additional space in the stack frame. Refer to *TPF C/C++ Language Support User’s Guide* for details on TMSPC and ICPLOG.

C Language Support Epilogs

The TMSEC and ICELOG macros are required for C library functions written in assembler and for entry points written in assembler that need to acquire a stack frame, like C written entry points. The TMSEC macro is for ISO-C functions and the ICELOG macro is for TARGET(TPF) functions. Each one deallocates the stack frame, restores the registers specified (or defaulted) by the prolog macro, and returns control to the calling function. There are two ways values can be returned, depending on their types: one way is to put the value in a return register, the other way is to put a pointer to the value in the parameter list.

Data Type to be Returned	ISO-C	TARGET(TPF)
Integer, Character, Pointer	RC=Rx	R6
`float`, `double`	1st fullword of C parameter list	1st fullword of C parameter list

The epilog macros can be coded anywhere in the program and any number of times. However, it makes good sense to code it once, immediately before the exit point of a function or program.

Secondary Linkage in ISO-C Function Libraries

All the functions in a library use secondary linkage if any one library function uses writable static or the library has active user exits. This results in performance losses for any function in the library. To keep these losses at a minimum, isolate the functions requiring writable static in less frequently used libraries rather than in libraries with frequently used functions and activate user exits only for libraries that require them.

Restrictions

Because library functions are not activated by way of the standard TPF ENTRC mechanism, there are certain considerations that must be kept in mind when coding a library function.

- Although it is not always a good idea, most SVC and fast-link TPF macros can be called from an assembler language library function. However, the TPF system services will not function correctly if general register 8 (R8) contains the base of the library function when the service is invoked; instead, the application program base must be reestablished before the macro call. This is done by loading R8 from ECB field CE1SVP. Before performing the load, the library function base should be stored in CSTKLBAS in the current stack frame so that it can be restored on return from the TPF service routine. In assembly programs that use ISO-C linkage when the LWS parameter is used with the TMSPC macro, storage for the function base is provided by the ILWSLBAS field. CSTKLBAS is always provided as long as the FRAMESIZE parameter is not NO. See the example that follows.

Coding Assembly Language Routines

Figure 37 shows the same library function written in assembler. The function calls the TIMEC macro to generate an 8-byte EBCDIC time stamp. The function returns the address of the time stamp to the calling C function.

```
*    ISO-C version                          TARGET(TPF) version
*                                |
   PRINT NOGEN                   |       PRINT NOGEN
   BEGIN NAME=C001,VERSION=41,   |       BEGIN NAME=C001,VERSION=31
          TPFISOC=YES            |
*                                |
   TMSPC FRAMESIZE=NO,LWS=R6     |       ICPLOG HIGHREG=R5,FRAMESIZE=NO
                                 |       L     R5,CE1TCA
                                 |       SL    R5,=A(CSTKTCA-ICS0TK)
   L     R5,ILWSUEXP             |       L     R5,CSTKUEXP-ICS0TK(,R5)
   ST    R8,ILWSLBAS             |       ST    R8,CSTKLBAS
   L     R8,CE1SVP               |       L     R8,CE1SVP
   TIMEC ,                       |       TIMEC ,
   L     R8,ILWSLBAS             |       L     R8,CSTKLBAS
                                 |       LR    R6,R5
   TMSEC RC=R5                   |       ICELOG ,
*                                |
   LTORG                         |       LTORG
   FINIS                         |       FINIS
   END                           |       END
```

Figure 37. Comparison of Sample Library Functions Written in Assembler

The blank areas in the ISO-C version of the sample program are just for spacing in this example and are not present in an actual program.

The BEGIN macro in the ISO-C versions uses the TPFISOC parameter because the default (that is, TPFISOC=NO) C language is TARGET(TPF).

The prolog macros, TMSPC and ICPLOG, are called to save the environment of the calling program. The TIMEC macro requires the use of R5 in this example. Saving the environment is automatic with ISO-C support. In TARGET(TPF) we must instruct ICPLOG to save registers R14–R5 using the HIGHREG parameter.

When the prolog macros finish, the assembly language program has addressability to a stack frame. For ISO-C the stack frame is addressed through the IDSDSA DSECT in R13. In TARGET(TPF) it is addressed through the ISC0TK DSECT in R7.

In either ISO-C or TARGET(TPF) if FRAMESIZE=NO, the stack frame addressed is that created for the program call. If FRAMESIZE is a number, the stack frame addressed is the newly created one. Such an additional stack frame is necessary to preserve the environment if the assembly language program calls another program or requires temporary storage. Library functions typically would specify FRAMESIZE=NO because they perform some simple service and return to their caller. Specifying FRAMESIZE=NO minimizes the amount of stack required and enhances performance slightly by avoiding unnecessary frame generation.

The ISO-C LWS parameter provides a simple means for accessing non-volatile work space. The LWS parameter provides a DSECT (IDSLWS) for accessing the structures in the work space. The address of the work space is returned, as specified in the example, in R6.

In ISO-C, if the CENV user exit is active, the user expansion area can be added to the initial stack and accessed through the ILWSUEXP data area. In the ISO-C example the address of the user expansion area is moved to R5 to begin setting up the call to TIMEC.

In TARGET(TPF), if the CSK user exit is active, addressability to the user expansion area can be set up in R5 through the TCA of the current stack and the CSTKUEXP. Storage in this area has been allocated by the C language support user exit (see "Customizing C/C++ Language Support" on page 301) and in this example is used to contain the time stamp returned by TIMEC.

Because TIMEC is a fast-link macro, R8 must contain the base of the calling application program when the macro is called. The library base address must be saved before the call to TIMEC. The ISO-C example uses ILWSLBAS for this because addressability to it is provided by the LWS parameter of TMSPC. If TMSPC had been called without the LWS parameter, the ISO-C example would have used CSTKLBAS with FRAMESIZE=0 to save the library base, just as the TARGET(TPF) example does.

On return from the TIMEC macro, the library function restores its program base from CSTKLBAS or ILWSLBAS, as appropriate.

When the TIMEC macro returns, in this example the address of the time stamp is in R5. In the ISO-C example, the TMSEC macro places the contents of R5 in R15 before the return. Therefore, when the assembly language program returns, the return value is a pointer to the time stamp.

Similarly, in the TARGET(TPF) example, the address of the time stamp is copied to R6, the register designated to contain function return values. A TARGET(TPF) ICELOG macro is called to return control to the calling C function in the application program segment.

User Expansion Area

The TPF system provides support for internal static variables defined with file or block scope. You may want to write library functions that provide their own static variable support, using the stack. There are two advantages to this: performance can be improved, and less storage is used. The user expansion area is provided in the first stack frame for this purpose.

There are 2 stack frame formats. The first stack frame of the first stack block is different from all the other stack frames that are eventually chained to it, which we will call subsequent stack blocks. The ICS0TK data macro maps the stack frames for TARGET(TPF) and the IDSDSA data macro maps the stack frames for ISO-C. (For more information about ICS0TK and IDSDSA, see their code listings.)

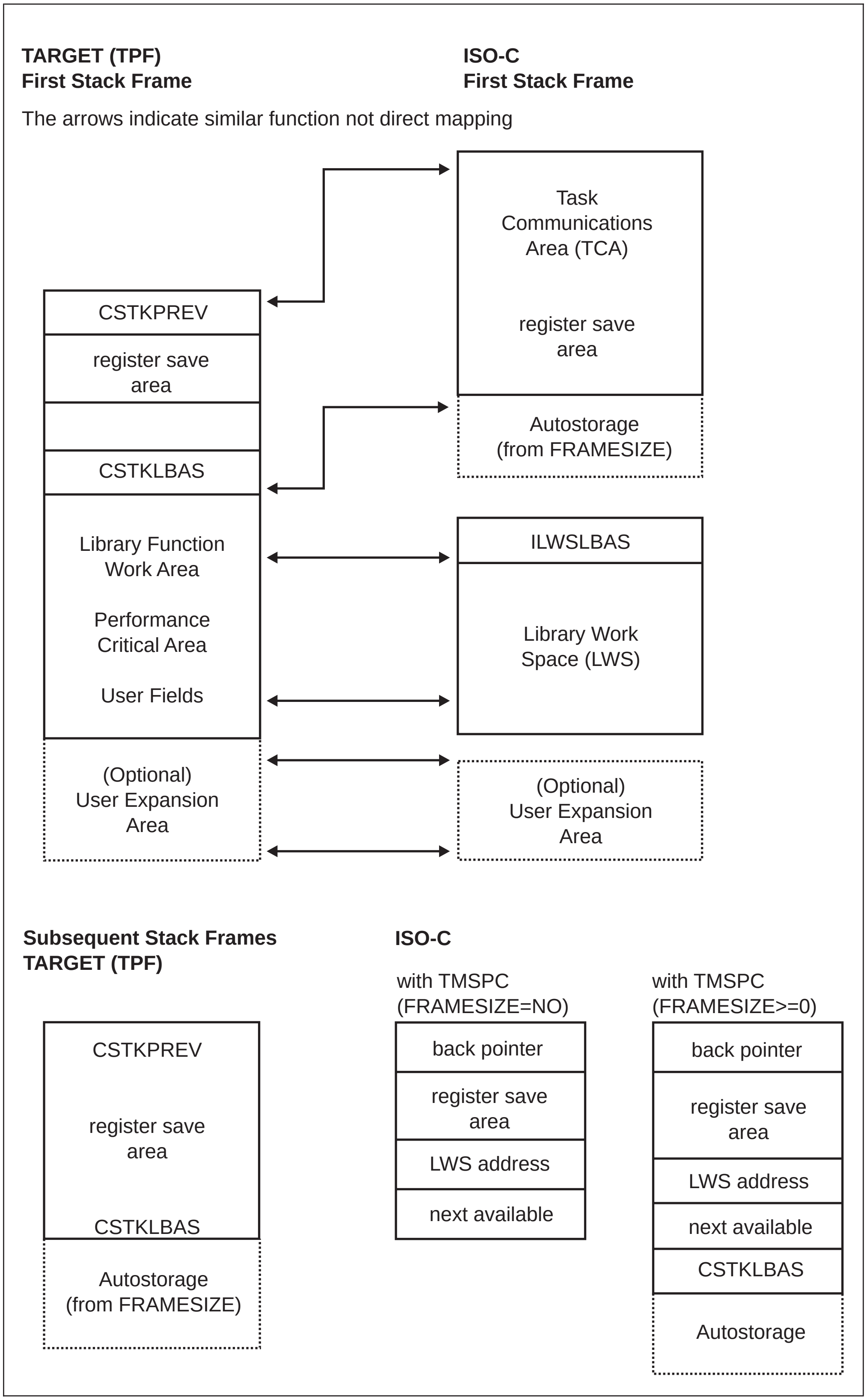


Figure 38. Stack Frame Formats

We talked about adding additional space to subsequent stack frames in “C Language Support Prologs” on page 220.

The area at the end of the first stack frame is known as the *user expansion area*. In ISO-C, the user expansion area is accessed through the LWS (DSECT IDSLWS) and in TARGET(TPF) it is accessed by the ICS0TK DSECT.

This area is available to application programs and is added during coding a user exit. A user exit allows TPF system users to add user-unique processing at various points in TPF programs.

The C stack exception routine user exit can be used to extend the first stack frame to include a user work area at the end. The name of the exit point is CENV in ISO-C and CSK in TARGET(TPF). The area itself is accessed with pointer CSTKUEXP in TARGET(TPF) and ILWSUEXP in ISO-C. It appears at the beginning of the library function work area.

See *TPF System Installation Support Reference* for more information about this particular exit and TPF user exits in general.

TPF Application Program Interface Functions

This chapter discusses macros and functions that perform similar operations. The discussion only mentions functions, but the same points are true also for macros unless specifically described otherwise.

Refer to “Language Structures and Case Guidelines” on page 1 for an understanding of why we provide both assembly language and C language structures in some cases and only C language structures in others.

The environment of an application program is the conceptual layer between the program and the TPF system. The environment is created when the system initially loads (IPLs) and its characteristics (such as the date, locale, libraries, and access to system resources) come from aspects of system operation. When the C library is brought into memory during restart, it is part of the environment and available to be shared by all C programs. The environment is implicit in each ECB by default even though the associated C structures are not physically allocated.

Environments are important because the ISO-C environment is different from the TARGET(TPF) environment. The TARGET(TPF) environment is composed of system state information a program can access from the TPF system. The ISO-C environment consists of state information provided through library functions and kept in the task communications area (TCA). TARGET(TPF) programs access TPF system structures to determine state information. ISO-C programs access state information in the TCA.

Transferring Processor Control

At times, several programs are required to process a single message. This can be done under the control of a single ECB, or the active ECB can create another independent entry, which has the effect of subdividing the processing. The active entry also has the ability to change the processing order by suspending its own processing or lowering its priority.

Enter/Back Services

Application programs transfer control to other entry points that use the same ECB with the enter/back services. The enter/back mechanism allows the application program to call or return to other application programs. The enter/back mechanism generates a code expansion that contains the information required by the linkage editor to create the linkage to reach the appropriate system service routine. The TPF system locates the programs in system storage and manages the main storage required to execute the programs. In assembler, only the 4-character program library name (for example, CYYA) is needed to call other programs. As programs return, main storage can be released by the system.

Information necessary to locate and execute an application program is transferred through one of the program Enter macros. Enter macros are coded explicitly by assembly language programs. The action of the ENTRC macro is performed automatically by the C language calling mechanism, so there is no C language equivalent of the ENTRC macro. There is a C function that corresponds to the ENTDC (Enter-Drop) macro. A request can be to:

- ENTRC: Enter a program with return expected.
- ENTNC: Enter a program with no return expected.

- ENTDC: Enter a program and release all other programs attached to the ECB (that is, programs that issued a prior ENTRC macro).
- BACKC: Return to a previous program (that is, to the last program to issue an ENTRC with no intervening ENTDCs).

The process of calling a chain of application programs under the control of the same ECB (using the enter/back services) is called *nesting*. The ECB contains 35 program levels that TPF uses to control program linkage. If more than 35 levels are required, TPF uses an additional block of working storage to control program linkages and correlate the ECB and the programs that it references. This linkage is transparent to the application program. However, the application programs must not exceed the nesting limit determined at system initialization. If the limit is exceeded, a system error will result and the entry will be exited.

You should release all programs as soon as possible to allow efficient use of main storage. This is the significance of the ENTDC macro or the `entdc` function. When the logic flow indicates that a series of programs no longer need to be activated, you can use `entdc` to release them. The ENTDC macro or the `entdc` function handles situations in which a programmer wishes to leave the C programming environment for the assembler environment. For ISO-C after a call to `entdc`, the stack pointer is reset to the top of the stack and the static storage is kept. For TARGET(TPF) after a call to `entdc`, all associated stack storage and static storage are released.

The C language programmer who also is experienced in assembler programming under TPF should note that the ENTNC assembler macro is not supported for C programming because a `goto` concept to an entry point does not exist in the C language.

Create Macros and Functions

Sometimes it is desirable to subdivide the processing of a given message to enable faster response to time-dependent functions or a more efficient program structure. TPF provides a set of macros and functions to permit an active entry to create another *independent* entry (TPF normally creates an entry as the result of an input message). When `tpf_cresc` is used, one creating entry can pass as much as 4 K bytes of data to a created entry and the created entry can pass a return value back to the creating entry. The creating entry can pass up to 104 bytes of parameter information to the created entry. Once TPF creates the entry, it retains the name of the creating entry at ECB location `ce1trc`, but that is the extent of any linkage between the 2 entries—they continue processing completely independently of one another.

TPF stores the parameters to be passed to the created entry in an interim main storage block that it adds as an item to a CPU loop list. This list is one of the following:

- Cross list —identifies entries for dispatching among I-stream engines
- Ready list—identifies items that are ready for immediate processing
- Input list—identifies items generated by messages from communication facilities
- Deferred list—identifies items that have been assigned low priority by an application program or by TPF for lack of resources.

TPF dispatches these lists, among others, whenever it is given control. The dispatching is in the above order; therefore, no item on the deferred list can be processed unless the input list is completely empty, and no item on the input list can be processed unless the ready list is completely empty.

When the item placed on the list by the create function is removed from the list for execution, the parameters to be passed to the newly created entry are moved into a new ECB that operational program zero (OPZERO) creates. An ECB is created for each new entry in the system and is assigned an activation number, which is obtained either from a system counter in OPZERO or from the ECB of the program executing the create function. The activation number determines which version of the E-type program the ECB will use. TPF then transfers control to the entry point specified in the create function call.

The type of create function called determines the priority of the processing that will be associated with the new ECB. Table 21 describes the create functions available to programmers using the C language under TPF.

Table 21. Description of Create Functions

Function	**Action**
`credc`	Creates a deferred entry and places the entry on the deferred list.
`creec`	Creates an entry with an attached main storage block, for immediate or deferred processing.
`cremc`	Creates an entry for immediate processing on the ready list.
`cretc`	Creates a time-initiated entry from an item placed on the ready list after a specified time interval. This function requires an interface with the system clocks.
`cretc_level`	Creates a time-initiated entry from an item placed on the ready list after a specified time interval. It places the entry on the specified core block reference word. This function requires an interface with the system clocks.
`crexc`	Creates a low priority deferred entry from the deferred list. This function is like `credc`, but `crexc` requires that more main storage blocks be available for real-time processing before it will execute. Programs that create many deferred entries use `crexc` to avoid depleting the number of blocks available for more critical real-time processing.
`sipcc`	In a loosely coupled environment, provides communication between processors; in a tightly coupled environment, provides communication between instruction streams. This is for system programs only.
`swisc_create`	Creates a new ECB on another I-stream.
`system`	Creates a synchronous entry for immediate processing of a program that has a `main` function and suspends the current entry.
`tpf_cresc`	Creates a synchronous entry for immediate processing on the ready list and suspends the current entry.
`tpf_fork`	Creates an asynchronous ECB on a specified I-stream.

All these functions subdivide the processing of a given entry by creating a new ECB and passing it to a specified entry point to continue processing. The entry point must have been allocated by means of the system allocator. The proper linkage also must be available to the entry point.

In assembly language, the program to be activated must be specified in the macro instruction operand. R14 and R15 must be used to reference the parameters being passed. For CREDC, CREEC, CREMC, and CREXC, R14 must be loaded with the number of bytes of parameters being passed, and R15 with the location of the parameters. For CRETC, R14 can be loaded with the time interval prior to activation or the time interval can be passed via the TIMEINC parameter of the CRETC macro. R15 contains data to be passed, or the PARM parameter can be used in the CRETC call. See the *TPF General Macros* for further details on using CRETC parameters.

The CRESC macro, which creates a synchronous entry for immediate processing, may request the creation of up to 50 ECBs. The ECBs are created in the same subsystem and subsystem user as the creating ECB (also called the parent ECB) but may be dispatched on any I-stream. The creating ECB may pass up to 4 K of data to each of the created ECBs. The creating ECB is placed on the wait list until the created ECB has completed. Upon exiting, the ECB that was created may pass a return value back to the creating ECB. See *TPF General Macros* for further details on using `tpf_cresc` parameters and register specifications.

Examples:

```
LA    R1,PGM             Name of program to enter
LA    R2,2               I-stream number
LA    R3,DATA            Address of data to pass
LA    R4,20              Number of bytes of data
LA    R5,100             100 second timeout
CRESC PROGRAM=(R1),WAIT=YES,DATA=(R3,R4),IS=(R2),TIMEOUT=(R5),
      RTNLST=(R6)
                         One ECB will be created and dispatched
                         on the I-stream specified by R2.  Twenty
                         bytes of data, beginning at DATA, will be
                         passed. The creating ECB will wait 100 seconds
                         for the created ECB to be completed
                         successfully.  Upon reactivation, the creating
                         ECB can access the value returned by the
                         created ECB using R6.


LA    R15,EBW020         Address of bytes to pass
CREMC ABCI               Activate program ABCI


LA    R14,48             Number of bytes passed
LA    R15,EBW020         Address of bytes to pass
CREMC ABCI               Activate program ABCI


LA    R14,10             Time increment value
L     R15,EBW040         Data meaningful to program ABCI

CRETC S,ABCI             Time interval is seconds (may
                         also be 'M' for minutes)
                         Activate program ABCI after 10
                         seconds

CRETC S,ABCI,TIMEINC=10,PARM=pgmlabel
                         Time interval will be placed in
                         R14 by the CRETC macro
                         Activate program ABCI after 10
                         seconds
                         Data meaningful to program ABCI at
                         label 'pgmlabel' will be placed in
                         R15 by the CRETC macro
```

You must supply the following arguments when calling C language functions `credc`, `cremc`, and `crexc`:

`length`
An integer between 0 and 104 that determines the number of bytes to be passed to the newly created ECB.

`parm`
A pointer to the parameters to be passed.

segname
A pointer to the entry point to be called.

A `creec` function call also must specify the data level that contains the working storage block to be passed to the new ECB. A `creec` function call also must specify an integer value, either `DEFERRED` or `IMMEDIATE` priority, to determine the list where the block is to be placed.

For `cretc`, you must specify the following arguments:

type
One of the predefined terms `MINUTES` or `SECONDS`, which indicate the units associated with the specified time interval.

segname
A pointer to the entry point to be called, as above.

units
An integer value that defines an increment of `MINUTES` or `SECONDS` that will elapse before the ECB is activated.

action
The address of 4 bytes to be passed to the created ECB. They will be copied into the work area of the new ECB `ebw000`–`ebw003`.

A `cretc_level` function call also must specify the data level of the core block reference word on which the ECB will be placed.

For `tpf_cresc`, you must specify the following arguments:

program
The name of the program that the created ECB will run.

istream
The number of the I-stream on which the created ECB will run.

wait
For WAIT=NO, initialize the created ECB; for WAIT=YES, create an ECB and dispatch all requested ECBs.

timeout
The number of seconds the creating ECB will wait for the created ECBs to be completed successfully.

rtnlst
The pointer that will contain the address of the event block with the return values of the created ECBs.

Examples of Using the Create Functions:

The following example creates a deferred entry for program COT0, passing the string “VPH” as input data to the program.

```
#include <tpfapi.h>
char *parmstring = "VPH";
 ⋮
credc(strlen(parmstring),parmstring,COT0);
```

The following example creates an ECB dispatched from the ready list for program OMA0, passing the string “755/15AUG” and the working storage block on level `D0` as input to the program.

```
#include <tpfapi.h>
char *parmstring = "755/15AUG";
 ⋮
creec(strlen(parmstring),parmstring,OMA0,D0,IMMEDIATE);
```

The following example creates an ECB dispatched from the ready list for program COT0, passing the string “VPH” as input data to the program.

```
#include <tpfapi.h>
char *parmstring = "VPH";
 ⋮
cremc(strlen(parmstring),parmstring,COT0);
```

The following example creates an ECB dispatched from the ready list for program QZZ0 after 5 seconds have elapsed. TPF will place the 4-byte character string “INIT” in `ebw000`–`ebw003` of the new ECB.

```
#include <tpfapi.h>
 ⋮
cretc(SECONDS,QZZ0,5,"INIT");
```

The following example creates a deferred entry for program COT0, passing the string “VPH” as input data to the program.

```
#include <tpfapi.h>
char *parmstring = "VPH";
 ⋮
crexc(strlen(parmstring),parmstring,COT0);
```

The following example creates two synchronous ECBs for immediate processing. The first ECB that is created will be passed 20 bytes of data and will be dispatched on I-stream 1. The second ECB that is created will not receive any data, will be dispatched on the main I-stream, and the creating ECB will wait indefinitely for the created ECBs to be completed.

```
#include <stdio.h>
#include <stdlib.h>
#include <tpfapi.h>

/* Initialize parameters for WAIT=NO call.                          */

char data[16] = "Hi..this is QXQZ";    /* Data to be passed         */

struct tpf_cresc_input tci;            /* Input structure for       */
                                       /* tpf_cresc()               */
struct tpf_ev0bk_list_data *ebptr;     /* Return value from         */
                                       /* tpf_cresc()               */

tci.program = "QPM1A";                 /* Pgm for child to enter    */
tci.istream = 1;                       /* I/S for child to run on   */
tci.wait = TPF_CRESC_WAIT_NO;          /* Don't create ECB yet      */
tci.data_length = 20;                  /* Pass 20 bytes of data     */
tci.data = data;                       /* Pass data at label DATA   */

/* Issue WAIT=NO call.                                              */

ebptr = tpf_cresc(&tci);              /* Issue WAIT=NO call        * /

/* Initialize parameters for WAIT=YES call.                         */
```

```
tci.program = "QPM1A";              /* Pgm for child to enter     */
tci.istream = TPF_CRESC_IS_MAIN;    /* I/S for child              */
tci.timeout = 0;                    /* Timeout value for parent   */
                                    /* Parent will wait forever   */
tci.wait = TPF_CRESC_WAIT_YES;      /* Create all children        */
tci.data_length = 0;                /* No data being passed       */
tci.data = NULL;                    /* No data being passed       */


/* Issue WAIT=YES call. All child ECBs requested to date will now   */
/* be created and dispatched.                                       */

ebptr = tpf_cresc(&tci);            /* Issue WAIT=YES call        */
```

Suspend Processing Macros and Functions

TPF provides 2 macros or functions that permit the entry currently using the CPU to request suspension of processing. TPF places suspended entries on either the input or deferred lists. This permits system programs to restore registers and return control to a suspended program.

The 2 functions used to suspend processing are:

Function	Action
`defrc`	Defer Processing of Current Entry — Places the entry at the bottom of the deferred list. The deferred list is simply the lowest priority list serviced by the CPU loop. After the entry is placed on the deferred list the system will service the higher priority lists.
`dlayc`	Delay Processing of Current Entry — Places the entry at the bottom of the input list: `dlayc` is, therefore, a shorter suspension than `defrc`.

No parameters are required with the `defrc` and `dlayc` functions.

The `defrc` and `dlayc` functions differ slightly from the `waitc` function. The `waitc` function is related to the completion of an I/O operation; if no I/O is pending then no suspension occurs. Both `dlayc` and `defrc` place the entry at the bottom of a list.

The period of time during which processing is suspended for deferred entries can be substantial. Accordingly, no records can be held by means of the record hold facility by an entry executing `defrc`, and no time dependent action (for example, response to a terminal) should follow this function. The entry should hold the least possible amount of working storage to minimize the impact on system performance.

Exit Functions

The exit functions are a special case of transferring control. They signify that processing of the entry is complete, and that the application wishes to release the remaining system resources and return control to TPF. Exit functions do the housekeeping required to remove an entry from the system. Resources held by the exiting entry are returned to the system for use by other entries.

When you call an exit function, the TPF system transfers control to the exit system service routine. Several library functions can cause this transfer of control to occur:

`exit`	Exits the ECB under normal circumstances.
`abort`	Exits the ECB under abnormal circumstances.

`serrc_op`	For C language, when coded with the predefined term `SERRCEXIT` as its first argument, calls the system error routine and then exits the ECB.
SERRC	For assembly language programs using SERRC (or its less expensive counterpart `snapc`), calls the system error routine and then exits the ECB.
`serrc_op_ext`	This function is similar to the `serrc_op` function with the addition that a prefix can be specified to identity the user or system program generating the dump.
`snapc`	Generates a snapshot dump and allows the program to stop or continue according to a parameter.

A program check (for example: an addressing exception, or an attempt to change protected areas of main storage) also causes the TPF system to transfer control to the exit system service routine.

Exit System Service Routine

The exit system service routine performs the following checks and functions when executing on behalf of `exit`, `abort`, `snapc`, and `serrc_op` with the `SERRCEXIT` option for C language or SERRC for assembly language:

- Passes control to any programs enrolled by the `atexit` function if the ECB exit was not abnormal (caused by a system error condition or by the `abort` function).
- Disconnects all programs associated with the entry from the ECB and decrements the demand counter for each disconnected program
- Releases stack and static storage
- Releases all frames used by the entry.
- Ensures that the ECB has not PAUSED the system. If a pause condition exists, the pause condition is ended.
- Clears any internal events associated with this ECB.
- Reactivates the creating entry if the ECB was created by a CRESC macro, `tpf_cresc` C function, or `system` C function.
- Unholds all records held by the entry.

  A held record means the entry was using the system conventions to prevent other entries from modifying the record.
- Closes all tapes or unit record devices still open to the entry
- Ensures that all I/O started by the entry is completed
- Passes control to the `exit` user exit (if active) to perform any user-defined exit services
- Returns the ECB to the available list.

The information needed for clearing an exited entry from the system is contained in various ECB fields.

Calling the Exit Functions

No operands are required with the EXITC macro. The parameters for the SERRC and `snapc` macros are described in *TPF General Macros*.

The exit functions for C language require the following arguments:

Function	**Arguments**

`exit`	An integer value that specifies a return code. In TARGET(TPF) if the value is nonzero, a system error dump bearing this number will be issued prior to exiting.
`abort`	None.
`serrc_op`	Four arguments specified in the following order: 1. A status argument (predefined to be `SERRCEXIT` for the purposes described above). 2. An integer of which the last 24 bits will be the identification number for the dump. 3. A pointer to the string to be displayed at the CRAS console and appended to the dump (or `NULL` if no message is desired). 4. A pointer to an array of pointers that indicates extra areas of storage to be displayed on the dump (or `NULL` if no storage list exists).
`serrc_op_ext`	The same 4 arguments specified in `serrc_op` and a prefix argument that associates an error with either an application or the IBM system.

Transfer Vectors

Transfer vectors are not supported for load modules. Load modules contain 1 external entry point.

In TARGET(TPF) C language transfer vectors allow a single program segment to have multiple entry points; in other words, transfer vectors allow a single program segment to contain multiple entry points. TPF handles calls to transfer vectors with the enter/back services: even when transfer vectors in the same program segment call each other. You should be aware of the general form of transfer vectors so that you will recognize them when you see them in existing code.

In TARGET(TPF) transfer vectors are coded into the system allocator and cross referenced to the program segment name. From C programs, transfer vectors can be called in the same way as normal entry points. If a C program contains multiple entry points, these functions are treated as transfer vectors. The IBM C compiler with TARGET(TPF) set automatically generates a branch table consisting of a series of unconditional branch instructions associated with each of the transfer vector names. Each transfer vector, once entered, functions as an independent entry point.

Main Storage Allocation

The TPF system and an ECB see virtual storage as 2 different address spaces: the system virtual memory (SVM) and the ECB virtual memory (EVM). The SVM contains all storage that can be used by a particular I-stream. The EVM contains all memory that can be referenced by an ECB. Each ECB has its own EVM. The layouts of both the SVM and EVM are shown in Figure 39 on page 237.

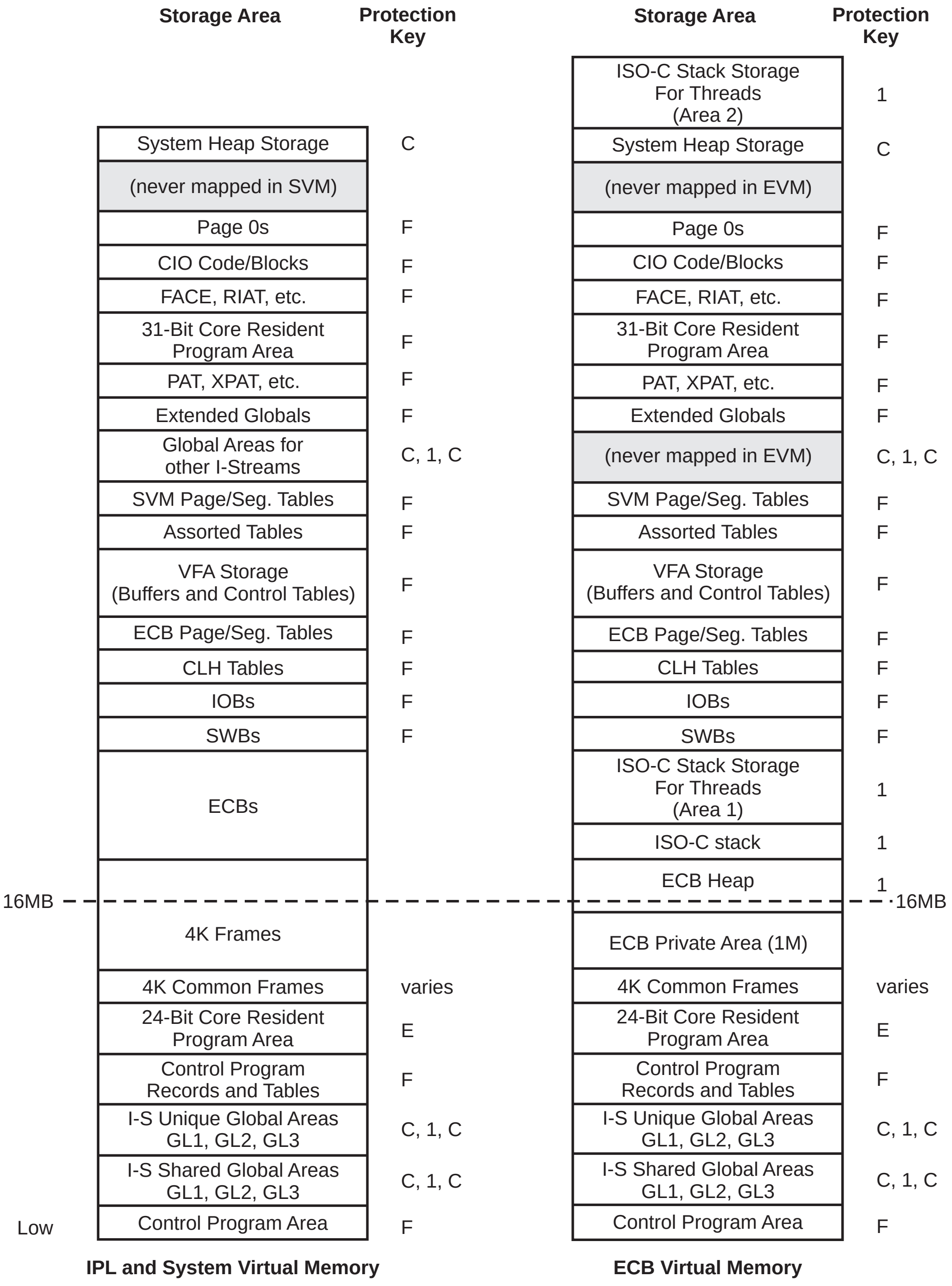
In a virtual storage system, main storage is almost synonymous with system virtual memory. Main storage is a critical resource that must be shared by the CP and ECBs. Its organization and efficient use is an important factor in a high performance system.

Part of main storage contains programs and data that are common to all entries and which are permanently resident there. This portion is called *fixed storage*, and is allocated at system generation. It typically includes at least:

- The control program and control program data
- Frequently accessed application programs and their required data, including the application global area

Another portion of main storage, called *working storage*, is allocated as 4 KB frames (in the SVM), from which working storage blocks are carved (in the EVM). Working storage is used to store and update data retrieved from file storage and consists of logical and physical blocks (which can be ECB private or shared) and contiguous or heap storage. (Logical block sizes are described in Figure 39 on page 237.)

It is the responsibility of system designers to allocate main storage areas to provide optimum system performance. The application programmer's concerns are simply to know how to access the fixed application data area and how to get and return working storage blocks.



Storage Area	Protection Key
System Heap Storage	C
(never mapped in SVM)	
Page 0s	F
CIO Code/Blocks	F
FACE, RIAT, etc.	F
31-Bit Core Resident Program Area	F
PAT, XPAT, etc.	F
Extended Globals	F
Global Areas for other I-Streams	C, 1, C
SVM Page/Seg. Tables	F
Assorted Tables	F
VFA Storage (Buffers and Control Tables)	F
ECB Page/Seg. Tables	F
CLH Tables	F
IOBs	F
SWBs	F
ECBs	
4K Frames	
4K Common Frames	varies
24-Bit Core Resident Program Area	E
Control Program Records and Tables	F
I-S Unique Global Areas GL1, GL2, GL3	C, 1, C
I-S Shared Global Areas GL1, GL2, GL3	C, 1, C
Control Program Area	F

IPL and System Virtual Memory

Storage Area	Protection Key
ISO-C Stack Storage For Threads (Area 2)	1
System Heap Storage	C
(never mapped in EVM)	
Page 0s	F
CIO Code/Blocks	F
FACE, RIAT, etc.	F
31-Bit Core Resident Program Area	F
PAT, XPAT, etc.	F
Extended Globals	F
(never mapped in EVM)	C, 1, C
SVM Page/Seg. Tables	F
Assorted Tables	F
VFA Storage (Buffers and Control Tables)	F
ECB Page/Seg. Tables	F
CLH Tables	F
IOBs	F
SWBs	F
ISO-C Stack Storage For Threads (Area 1)	1
ISO-C stack	1
ECB Heap	1
ECB Private Area (1M)	
4K Common Frames	varies
24-Bit Core Resident Program Area	E
Control Program Records and Tables	F
I-S Unique Global Areas GL1, GL2, GL3	C, 1, C
I-S Shared Global Areas GL1, GL2, GL3	C, 1, C
Control Program Area	F

ECB Virtual Memory

Figure 39. Virtual Storage Layout

Note: TPF's virtual address spaces makes it impossible to use a full 2 gigabytes of real memory.

Allocating Working Storage

Working storage is main storage that is allocated from 4-KB frames and attached to entries during their system life. The elements allocated, for example, are used for the following:

- Entry control blocks (ECBs)
- Data event control blocks (DECBs)

- Programs required from file storage
- Data blocks either created in main storage or transferred from file storage.

The TPF system allocates storage for automatic variables and parameter lists in C language every time you call an external or library function. Much of this is transparent to the application. However, your application program also may need working storage blocks for its own processing. You can reference a limited number of working storage blocks in your application program from fixed locations in the ECB or DECB.

You obtain access to a single working storage block with the `getcc` function, which places the address of the block on a specified ECB data level or DECB. You can release working storage blocks with the `relcc` function.

You can obtain access to DECBs by using the `tpf_decb` types of functions or by using the DECBC macro. For more information about DECBs, see “Data Event Control Blocks” on page 29. For more information about the `tpf_decb` types of functions or the DECBC macro, see the *TPF C/C++ Language Support User’s Guide* and *TPF General Macros*, respectively.

You can obtain access to a heap storage block with the `malloc` function and release the storage with the `freec` function. The argument used with the `relcc` function and the two arguments used with the `getcc` function are an enumerated data type that is defined in the `tpfapi.h` header. Examples follow.

The following is an example for assembly language.

```
GETCC D3,L0
```

This statement requests a 127-byte block on data level 3. TPF system programs will assign one main storage address from the 127-byte pool to the ECB, and will update the CBRW at level 3:

	Core address	Block type held	Bytes in block
CBRW Level 3	0003A420	0011	007F

On return from the GETCC macro, register 14 (R14) will also contain the main storage address of the block, which the ECB can now access in the ECB private area. Of course, the ECB must not be holding a block at the specified level when the macro is issued. If it is, a system error will occur and the ECB will be forcibly exited.

The application can also request a 127-byte common block to share with other ECBs using the COMMON= parameter, for example:

```
GETCC D3,L0,COMMON=YES
```

The ECB accesses common blocks in the ECB common area (assuming it is properly authorized).

The release main storage macro is used to release blocks, one at a time, when they are no longer needed. For example:

```
RELCC D3
```

The block size is not required. TPF will return it to the pool and update the CBRW. After the RELCC request, the level 3 CBRW (see above, after the GETCC) will appear as follows:

CBRW Level 3	0003A420	0001	007F
	Core address	Block type held	Bytes in block

Note that only the block type held indicator is modified. The remainder of the CBRW is unchanged. For additional information on use of the CBRW, see “Core Block Reference Words” on page 28.

When using RELCC, a block must be held at the specified level; if not, a system error will occur and the ECB forcibly exited.

The following is a C language example.

```
#include <tpfapi.h>
#include <c$am0sg.h>
struct am0sg *amsg;              /* pointer to message block */
⋮
amsg = getcc(D3,GETCCTYPE,L1);  /* attaches a 127-byte block on data
```

TPF system programs will assign one main storage address from the 127-byte block pool to the ECB, and will update the CBRW at level 3:

CBRW Level 3	0003A420	0011	007F
	Core address	Block type held	Bytes in block

The `getcc` function normally returns a pointer that represents the starting address of the newly obtained working storage block, which the ECB can now access in the ECB private area. Of course, the ECB must not be holding a block at the specified level when the function is called. If so, a system error will occur and the ECB will be forcibly exited.

The `relcc` function is used to release blocks, one at a time, when they are no longer needed. For example:

```
#include <tpfapi.h>
⋮
relcc(D3);                       /* releases the storage block on data lev
```

The block size is not required. TPF will return the block to the pool and update the CBRW. After the `relcc` request, the level 3 CBRW (see above, after the `getcc` function call) will appear as follows:

CBRW Level 3	0003A420	0001	007F
	Core address	Block type held	Bytes in block

Note that only the block type held indicator is modified. The remainder of the CBRW is unchanged. For additional information on use of the CBRW, see “Core Block Reference Words” on page 28.

When using `relcc`, a block must be held at the specified level; if not, a system error will occur and the ECB will be forcibly exited.

Assembly language and C language applications can also request (and release) storage in blocks of varying sizes (greater than 4 KB) from contiguous or heap storage using the MALOC or `malloc`, CALOC or `calloc`, RALOC or `realloc`, and FREEC or `free` functions. This storage resides in the heap private area, above 16 megabytes.

If the ECB exits with main storage blocks being held, TPF releases them. ***This is generally the most efficient way to release this storage,*** assuming that an ordinary amount of system resources are being used.

In the case of reading from and writing to *file* storage, it is TPF (rather than the application) that manages the working storage blocks. For a file read, TPF gets a block of the proper size, then reads in the file record. For a file write, TPF writes, and may release, the block. A system error will occur if the application attempts a file read into an occupied data level, or attempts to release a block after a file write (assuming that the write was not performed on behalf of a file with no release function).

TPF provides additional functions that combine the allocation of main storage blocks and file retrieval and storage. See “Summary of File Reference Functions and Macros” on page 260, the *TPF C/C++ Language Support User’s Guide*, *TPF General Macros*, and *TPF System Macros* for more information.

Application Global Area

The fixed data area called the *application global area* is of prime importance to the application programmer. Although in practice the TPF system also uses the application global area, it is primarily an application area. The purpose of the area is to provide efficient access for all programs to commonly and often highly accessed data. A detailed discussion of the concepts and format of the global area is given in the *TPF System Installation Support Reference*.

Figure 40 on page 241 shows a layout of the application global area. Conceptually, global area 1 and global area 3 are similarly structured protected areas. Each contains a directory, records for common system and application values, and other protected data records. Global area 2 contains nonprotected data records. An extended global area, residing above the 16-MB boundary, also can be specified; its inclusion in the system is optional. The structure of this area is analogous to the primary area. Application programs can take advantage of the greatly extended storage capacities of 31-bit addressing mode, at the same time relieving storage constraints below the 16-MB boundary. Support is provided under TPF for writing application programs in the C language that interact with global areas 1, 2, 3, and their extended areas.

It is important to note that the TPF global areas are used by programs written in both assembler and the C language and that ***different naming conventions apply***.

TPF convention requires that all symbolic names for the global area start with the @ character in assembly language and _ in C language. The remaining characters in the symbol are in lowercase.

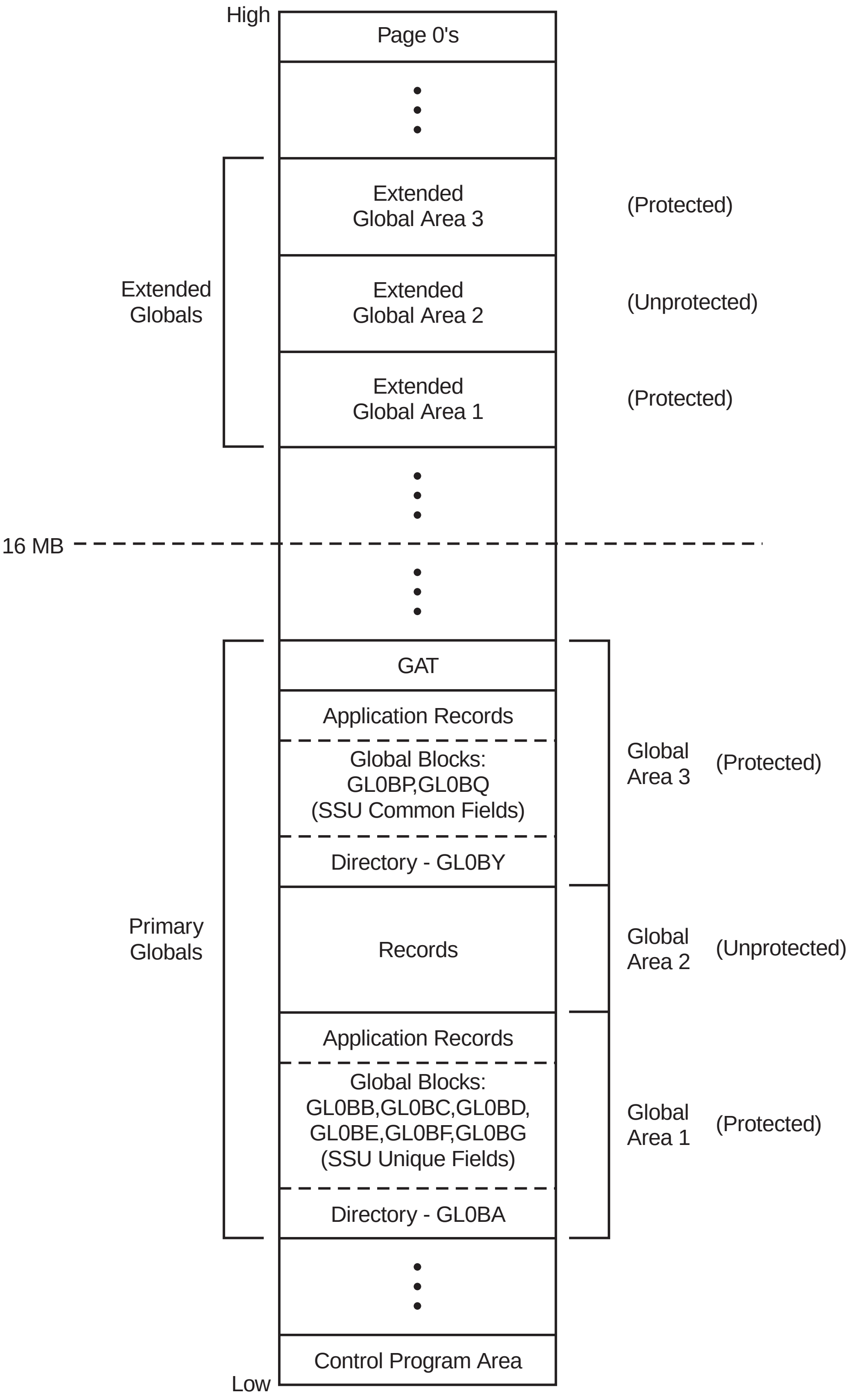


Figure 40. Global Storage Allocation for a TPF Basic Subsystem with a Single I-Stream. Terms labeled GL0xx correspond to the names of assembler DSECTs associated with the areas.

Global Directory

A global directory (@GLOBA for global area 1 and @GLOBY for global area 3) is simply a series of pointers, each with a symbolic name, containing the main storage address of each record or record type in the global area.

For assembly language, the GLOBZ macro provides a DSECT and direct addressability via symbolic names for the first 4096 bytes of global area 1 and global area 3. The directory and the common system/application value records must reside in the 4096-byte limit, but the protected resident records need not.

If the item pointed to is a multiple record file, then the pointer refers to the first record of the file. The application must then develop its own logic for stepping through the file. Because global records must be backed up in file storage, the directory item also contains the system file address.

Common Values

Immediately following each directory is a variable number of records that contain the common system and application values. Each field in these records has a unique symbolic name and either can be operated on or indirectly addressed. In assembly language the GLOBZ macro is used. For example, @CFLTN can be a symbolic name for a 4-byte field assigned in a miscellaneous common value global record. After issuing GLOBZ, the program can address @CFLTN with any standard instruction.

Protected Data Records

The protected resident records are pointed to by the directory (@GLOBA), but fields in these records are ***not*** defined by GLOBZ. Each resident record type has its own data macro, and the assembly language program addresses it by loading a base register with the pointer from the directory.

C language access to the protected data records in global areas 1 and 3 may be obtained only by means of a pointer returned by the `glob` function, (see “Global Area Functions for C Language” on page 246). It is the application programmer’s responsibility to write a structure to examine the contents of the record itself.

For example, assume that one of the protected resident records is an exception authorization record that has its fields defined in a data macro named EX0AU, and has a pointer in the global directory labeled @ or _. Your program could address the fields in the exception record with the following code:

```
GLOBZ REGR=R1            Defines global fields and loads base register
EX0AU REG=R2             Assigns register 2 as base for exception record
L     R2,@EXAU           Loads exception record base with pointer from
                           global directory
CLC   EXAU1,EBW020       Example instruction using detail field from
                           exception record data macro
```

Note: Your program should not give up control of the processor (by issuing any I/O request, such as FINWC) between reading a global record or field and updating that record or field. If it does, the data that your program reads may have already been updated by another program by the time your program decides to update it.

Maintaining Global Areas

Global areas are maintained either:
- From common main storage, or
- Across multiple copies of main storage or DASD.

This organization affects the techniques that are used when updating the global areas.

When global data in a central processing complex (CPC) exists in common storage, application programs must use appropriate multiprocessing techniques. See the *370/XA Principles of Operation*.

When global data must be maintained identically across multiple copies, global synchronization services should be considered. A detailed discussion is given below.

Synchronizing Global Areas

Global synchronization provides an application programmer with a method of coordinating values in main storage among several active I-streams. This need arises when multiple I-streams (processors) in a multiple processor CPC and/or a loosely coupled (LC) complex use a shared global resource. All I-streams in the complex share DASD and have access to all data that reside on that DASD.

In a Multiple Data Base Function system, a global area is defined for each subsystem user (SSU). A global area also is defined for each processor in a CPC and in an LC complex generated with the High Performance Option licensed feature.

Loosely coupled systems allow terminals to be connected to several CPCs, all of which use the same database. Although most of the data required for an input message to be processed exist on shared DASD storage, some of the critical data resides in global main storage. This data must be maintained (as much as possible) concurrently in all processors connected to the network. In other words, the data must be *synchronized*.

Each program that relies on the most current value of data in a global area must ensure the presence of the most current data in main storage by using a SYNCC synchronization macro if in assembly language, a `GLOBALSYNC` argument with the `global` function if in C language, or if in ISO-C the `glob_update` or `glob_sync` functions. (See the “Global Area Functions for C Language” on page 246, “Synchronization Considerations” on page 249 and the description of the global function or the SYNCC macro in *TPF General Macros* for more information.)

Keypointing Global Areas

In TPF, certain critical data in main storage is backed up on file. This data is maintained in records called *keypoints*, and the process of backing them up is called *keypointing*. Some global data, when it is modified, is not keypointable. All other data ***must*** be keypointed whenever it is modified. A C language application programmer using the `global` function in ISO-C or TARGET(TPF), or `glob_keypoint` in ISO-C has the option of specifying that keypointing should take place or allowing the decision of whether or not to keypoint data to be handled by the function itself. More details are given in “Operation of the Global Functions” on page 246.

Global Area Macros for Assembly Language

The use of global area macros varies greatly depending on whether you are accessing synchronized or nonsynchronized global records and fields.

GLOBZ This macro is used for both synchronized and nonsynchronized records and fields.

GLOBZ is an executive macro as well as a data macro. It defines the symbolic labels to be used in addressing data in the first 4096 bytes of global areas 1 and 3. It also loads the base register specified in the instruction with the address of the global area requested. It has options for global area 1 or global area 3 or both, and to specify a field name to be accessed when the global area is not known.

```
Examples: GLOBZ REGR=R1                  Loads R1 with base of
                                        global area 1
  GLOBZ REGS=R2                          Loads R2 with base
                                        of global area 3
  GLOBZ REGR=R1,FIELD=@CFLTN             Loads R1 with base of
                                        global area containing
                                        the field @CFLTN
```

GLMOD This macro used only for nonsynchronized records and fields.

Application global areas 1 and 3 are in protected main storage. Before they can be modified the application program must first issue the GLMOD macro to change the protection key. Global area 1 or global area 3 may be specified as an operand; global area 1 is the default.

FILKW This macro is used only for nonsynchronized records and fields.

After the global area is modified, the application program must restore the protection key. Until it is restored, the application cannot modify its own ECB or other standard data areas. This is done with a parameter of the file keyword macro, FILKW.

FILKW can also request that critical main storage records be backed up on file. In TPF, these records are called *keypoints*, and the process of backing them up is called *keypointing*. Some global records—those that are not modified—are not keypointable. All others ***must*** be keypointed whenever they are modified.

FILKW parameters are:

- Restore or don't restore the protection key.
- The symbolic name of the field containing the address of the global record to be keypointed or a field in the global record to be keypointed. (Note that if the field parameter is used, the specified field must be coded in a table contained in the FILKW code itself.)

Up to 8 records or fields may be specified in one FILKW statement.

Examples:

```
FILKW R,@GBLCC                        Restore protection key
                                      and keypoint record
                                      @GBLCC

FILKW N,@GBLCC,@GLOBA                 Update keypoint records
                                      specified; do not restore
                                      protection key.

FILKW R,@GLOBA,@GLBCC,@CFLTN,FLD=YES   Restore protection key
                                      and keypoint specified
                                      records. At least 1 of
                                      the names specified is
                                      a field in a record.
```

SYNCC This macro is used only for synchronized global records and fields.

The SYNCC macro is used to (1) gain exclusive use of a synchronizable global field to update it and (2) notify other processors in the multiple processor complex that the updates have been made. The synchronization macro and its supporting logic in the control program ensure that the most current data is obtained from the global area, and that any updates to a global record or field are refreshed in the main storage of all the loosely-coupled processors.

SYNCC options are:

- LOCK—Retrieve the most recent file copy of the data and move the contents of the field or record into main storage at the appropriate location. An additional function of the LOCK option is to maintain a hold on the file copy to prevent simultaneous updates to the file copy. This results in a hold in the processor’s record hold table and a hold at the disk control unit. The LOCK option will, upon return to the calling program, set the protection key to the proper value for the global field or global record requested.
- UNLOCK—Release the file record after a LOCK has been issued, but no update (and therefore no synchronization) is required. The UNLOCK option restores the protection key to the proper working storage key.
- SYNC—Refresh the main storage images of modified data in every active processor by (1) copying the data to file storage; (2) informing the other loosely-coupled processors, via a facility called interprocessor communication (IPC), that the data should be immediately refreshed from that file copy; and (3) informing the other processors in the CPC that their copy of the data must be updated. The SYNC option, like the UNLOCK option, restores the protection key to the proper working storage key and releases the hold on the file copy.

The order and timing of the options you specify on the SYNCC macro is critical to the operation of your program.

- You must specify a LOCK option before either a SYNC or UNLOCK option; otherwise, your program will be exited. Furthermore, if any input or output requests are pending when you specify SYNCC LOCK, your program will be exited.
- Your application program must process synchronized fields and records 1 at a time. This means each LOCK option must be followed by SYNC or UNLOCK before another LOCK can be issued.
- The program segment that issues a LOCK option must also issue the SYNC or UNLOCK option before entering another segment or issuing an EXITC macro.

In a loosely-coupled environment the SYNCC macro results in a find-wait-hold on the file record that services the synchronized global field or record. As a result, it prevents other programs on the same processor or any other active processor from accessing the referenced global field or record. To shorten the duration of this ″lockout″, you should keep the number of instructions executed between the LOCK and the SYNC or UNLOCK options to the minimum number required to perform the updates.

The SYNCC macro issues a supervisor call to refresh main storage from file storage. This is a slow I/O process during which the program that issues the SYNCC does not have control.

Special Coding Considerations

Because the SYNCC macro modifies the protection key, data in working storage cannot be manipulated after SYNCC LOCK has been issued. If you need access to working storage before all global updates are complete, you can use FILKW and GLMOD macros to alter protection keys appropriately. For an example, see the application global area in the *TPF System Installation Support Reference*.

Global Area Functions for C Language

Before considering the operation of the global functions, an application programmer should follow the programming considerations that apply to using the global functions provided with IBM C language support. An application programmer who intends to access global data in the extended global areas should follow the addressing conventions for those areas. After considering the operation of the global functions, the programmer who intends to employ synchronization techniques should regard the special treatment that applies.

Each of these topics is discussed in more detail below.

Programming Considerations with the Global Functions

It is important to note that the TPF global areas are used by programs written in both assembler and the C language and that ***different naming conventions apply***. TPF convention requires that all symbolic names coded in assembler for the global area begin with the @ character. The C language prohibits the @ character in any identifier; TPF convention calls for an underscore (_) to be used in its place. All remaining characters are specified by convention in lowercase.

Support for the C language requires that all names used to refer to global fields and records be previously defined in the `c$globz.h` header and also requires that the `tpfglbl.h` header, which includes `c$globz.h`, be included in any code that calls the global functions. Each global tag name in the `c$globz.h` header is assigned a unique 32-bit number that describes:

- The global item's displacement in the global area of residence
- Its length as defined
- The number of the global area in which it resides
- Characteristics of online handling, such as keypointability and subsystem commonality or uniqueness.

The `c$globz.h` header file normally is created at installation time and must be revised every time that the definition or order of items in the global areas change. Because the compiler treats the contents of `c$globz.h` as constant values, existing C programs that reference global tag names must be recompiled whenever the `c$globz.h` header is revised.

Sample code that may be helpful in generating the contents of `c$globz.h` has been provided with the installation tapes. A more complete description of the use of this code is given in the *TPF C/C++ Language Support User's Guide*.

Operation of the Global Functions

TPF provides 2 C functions for ISO-C or TARGET(TPF) for the application programmer who wants to access data in the global areas: `glob` and `global`. The use of global area functions varies greatly depending on whether you are accessing a global record or field. You can use the `global` function only with a global field and only to operate on that field, which is discussed in more detail below. You can use the `glob` function with either a global field or record solely to obtain a pointer to the global data. In the case of a global field, `glob` returns the field address. In the case of a global record, `glob` returns the record's directory entry address (core/file) indicated by the tag.

Note: A program should not give up control (by calling an I/O function, such as `finwc`) between reading and updating a global field. If it does, the data that the program reads may have already been updated by another program by the time the program decides to update it. Use of the locking facility provides additional security when updating synchronizable global fields.

The operation of the `global` function is determined by the action that is specified as the second argument in the function call. The predefined terms that can be used to specify the global operation to be performed include:

Term	Action
`GLOBALUPDT`	Updates the referenced global field with the value pointed to by a third argument that is required in the function call. Additional operations (keypointing, synchronizing) determined by the field's attributes are performed on the referenced global.
`GLOBALMODF`	Alters the referenced global field with the value pointed to by a third argument required in the function call. No additional keypointing or synchronization are performed on the field.
`GLOBALCOPY`	Copies the contents of the referenced global field to the place in automatic storage pointed to by a third argument required in the function call.
`GLOBALKEY`	Causes keypointing of the referenced global field. A third argument is not required with `GLOBALKEY`.
`GLOBALLOCK`	Reserves the exclusive use of the referenced global field and forces its main storage copy to be refreshed from the most current file copy. The contents of the field then are copied to automatic storage. The `GLOBALLOCK` operation maintains a hold on the file copy to prevent simultaneous alterations to the file copy. This results in a hold in the processor's record hold table and a hold at the associated disk control unit. Programs that use `GLOBALLOCK` should use `GLOBALUNLK` or `GLOBALSYNC` as soon as possible to prevent severe system degradation caused when other ECBs are awaiting access to the referenced global field.
`GLOBALUNLK`	Releases exclusive use of a referenced global field without altering it. A third argument is not required with `GLOBALUNLK`.
`GLOBALSYNC`	Releases exclusive use of a referenced global field and synchronizes the global values across processors in an LC complex. A third argument is not required with `GLOBALSYNC`. The `GLOBALSYNC` operation is used to (1) gain exclusive use of a synchronizable global field to update it and (2) notify other processors in a multiple processor complex that the updates have been made. The synchronization operation and its supporting logic in TPF ensure that the most current data is obtained from the global area, and that any updates to a global field are refreshed in the main storage of all the LC processors. The `GLOBALSYNC` operation refreshes the main storage images of modified data in every active processor by (1) copying the data to file storage; (2) informing the other LC processors, via a facility

called interprocessor communication (IPC), that the data should be immediately refreshed from that file copy; and (3) informing the other processors in the CPC that their copy of the data must be updated. The SYNC operation releases the hold on the file copy.

ISO-C support provides for manipulating global fields and records with the `glob` function. The 6 operations available are similar to those available with the `global` function.

Term	Action
`glob_keypoint`	Keypoint a global field or record This function causes the keypointing of the specified TPF global field or record. This function performs the equivalent of a GLOUC assembler macro.
`glob_lock`	Lock and access a synchronizable global field or record This function reserves exclusive write access to the specified TPF global field or record and forces refreshing of the core copy from the most current file copy. This function performs the equivalent of a SYNCC LOCK assembler macro.
`glob_modify`	Modify a global field or record This function copies the modification data to the specified TPF global field or record.
`glob_sync`	Synchronize a global field or record This function releases the exclusive use of the specified TPF global field or record, and synchronizes the field or record across processors in a loosely coupled complex. This function performs the equivalent of a SYNCC SYNC assembler macro.
`glob_unlock`	Unlock a global field or record This function releases the exclusive use of the specified TPF global field or record. This function performs the equivalent of a SYNCC UNLOCK assembler macro.
`glob_update`	Update a global field or record This function updates data contained in the specified TPF global field or record. If the tag name is synchronizable, the equivalent of a SYNCC SYNC assembler macro is performed to synchronize the global area across the loosely coupled complex; otherwise, if the tagname is keypointable, the equivalent of a GLOUC assembler macro is performed to keypoint the global record specified by tagname (or the record containing the global field).

Synchronization Considerations

The order and timing of the operation you specify for synchronizing a global field is critical to the operation of your program. These considerations are true for assembly language programs, TARGET(TPF) functions, or ISO-C function.

- You must specify a locking operation (`global GLOBALLOCK` or `glob_lock`) before either a synchronization (`global GLOBALSYNC` or `glob_sync`) or unlocking operation (`global GLOBALUNLK` or `glob_unlock`); otherwise, a system error results. Similarly, if any I/O requests are pending when you specify a locking operation (`global GLOBALLOCK` or `glob_lock`), a system error results.

  The `global GLOBALUPDT` and `glob_update` functions automatically synchronize the field or record being updated, provided the synchronization indicator is set. The record or field should be locked prior to calling the update function.
- Your application program must process synchronized fields 1 at a time. Each locking operation (`global GLOBALLOCK` or `glob_lock`) must be followed by a synchronizing (`global GLOBALSYNC` or `glob_sync`) or unlocking (`global GLOBALUNLK` or `glob_unlock`) operation before another locking operation can be called.
- External functions that call the locking functions must also call the synchronizing or unlocking functions before calling another external or library function.

The locking functions (`global GLOBALLOCK` and `glob_lock`) result in a find-wait-and-hold (see “File Storage Access” on page 257) on the file record that services the synchronized global field. As a result, if any other entries on the same I-stream or any other active I-stream in the loosely-coupled complex attempt to simultaneously lock the same global field, their request will be queued. Such entries will be able to access the specified global field only after the first entry has released its lock through either an unlocking function (`global GLOBALUNLK` or `glob_unlock`) or a synchronizing function (`global GLOBALSYNC` or `glob_sync`).

At any given time only 1 ECB may hold the lock for a particular global field. To shorten the duration of this “lockout”, keep the number of instructions executed between the locking functions and either the synchronizing or unlocking functions to the minimum number required to perform the updates.

The synchronizing functions (`global GLOBALSYNC` or `glob_sync`) issue a supervisor call to refresh main storage from file storage. This is a slow I/O process during which the program that called for the synchronization does not have control.

Examples of Using the Global Functions

In the illustrative process flow introducing the application interfaces, our credit application stored its floor limit, an integer, assigned the symbolic name `CFLTN` in a global field. It was updated with the value pointed to by `amount` as follows:

```
#include <tpfglbl.h>
⋮
global(CFLTN, GLOBALUPDT, &amount);
```

If a protected resident record in the extended area were an exception authorization record that had its fields defined in a structure named `ex0au`, and had a pointer in the global directory labeled `_exau`, a program could examine the fields in the exception record with the following code:

```
#include <tpfglbl.h>
#include <tpfapi.h>
struct ex0au {
  ⋮
            } *@EXAU;
  ⋮
 @EXAU = *(struct ex0au **) glob(_exau);
  ⋮
```

It would be up to the application to know and properly define the structure used above.

Accessing Data in Assembly Language and C Language

Database support is critical to the performance of any system. Performance of a system is largely dependent on the number of file storage requests and the time required to transfer them. TPF is designed to optimize the queuing and transfer time, but the number of requests is largely determined by application design. Therefore, good performance in any online system is the result of TPF support facilities, plus good application design and use of those facilities. For this reason, it is important that application programmers understand the concepts and basic structure supported by TPF.

High performance is an assumed objective of TPF data support. This implies 2 secondary objectives:

1. The online files are a common data resource for all users and applications. Applications may be designed to provide independent function, but all share the same data resource. The application may structure data to its unique requirements, but the structuring must be in the context of the basic common data structure. This common database is allocated and structured at system generation. There is no facility comparable to MVS jobs or JCL which allow file allocation per job. The program or entry must use the preallocated system data resource. (See “General Data Set and General File Support” on page 273 for a discussion of MVS-type data sets in TPF.)
2. File addressing is symbolic so that the application programming is independent of the physical configuration. Essentially, all disk files are organized for direct access and are allocated across the entire physical storage to improve performance. The application always deals with a symbolic reference number that system programs and tables translate ultimately to a physical address. The user configuration can expand, take on new devices or eliminate old, without requiring application programming changes.

The main database of a TPF system comprises 2 categories of disk data file: the fixed file and the random pool. (General data sets and general files are excluded from this discussion. See “General Data Set and General File Support” on page 273.) Programs that run on the TPF system are another category of disk file, although they are not considered data.

Fixed File

The fixed file is analogous to a conventional multivolume data set organized for direct access. The storage area in the fixed file is allocated at system generation to specific functional application record types. An application cannot create a new record type, nor can it add to the allocation for a record type, during execution. It can use only what was allocated during system generation. The characteristics of the fixed file determine the type of records that should be placed in the fixed file.

- Because the number of records can be changed only by regenerating the system, the records required by applications should be fairly constant in number—otherwise, the file space will not be well used.

- Because the fixed file ordinal number provides an accessing scheme for direct retrieval, it is appropriate to place frequently accessed records there.

For example, the fixed file often is used to contain records that are pointers to data in the random pool file area (see “Random Pool File Area” on page 254).

A given record type may be allocated on the fixed file as small (381 bytes), large (1055 bytes), or 4 K (4095 bytes). For assembly language, the record types are assigned symbolic names (which begin, by convention, with a #) and are equated in the system equate macro (SYSEQC) to an absolute value. For C language, the record types are assigned symbolic names and equated to absolute values in `c$syseq.h` using `#define` preprocessor statements.

For more information about fixed file storage, see *TPF System Generation*.

FACE, FACS, and FAC8C

In a record type, each record has a record ordinal number (also called relative record number or RRN). However, to retrieve a record, the TPF system must know the ordinal number of the record across the entire DASD base. The file address compute programs (FACE and FACS) and the FAC8C macro calculate this number from the record type and record ordinal number.

FACE, FACS, and the FAC8C macro each consist of a system utility program and associated index table residing permanently in main storage. An application programmer must know the record type of the record to be retrieved. The application must develop the algorithm that calculates the desired ordinal number in that record type.

Figure 41 on page 252 shows an example of a simple fixed file structure with 4 record types, the FACE/FACS table, and SYSEQC items that define it. Usually the numeric record type is set up using an equate or a `#define`. Nevertheless, any changes to the FACE index table may require programs that use FACE to be recompiled.

The FACE and FACS programs and the FAC8C macro are similar:
- FACE is passed numeric record types in the form of equates when coding in assembler language or `#define` statements when coding in C language.
- FACS is passed symbolic record types.
- FAC8C is passed numeric or symbolic record types.

The numeric record types may be changed from time to time in the FACE table; they may also vary from MDBF subsystem to subsystem. This means programs that use FACE must be recompiled when the FACE table is changed or when they are transported between systems. The symbolic record types used by FACS avoid this drawback.

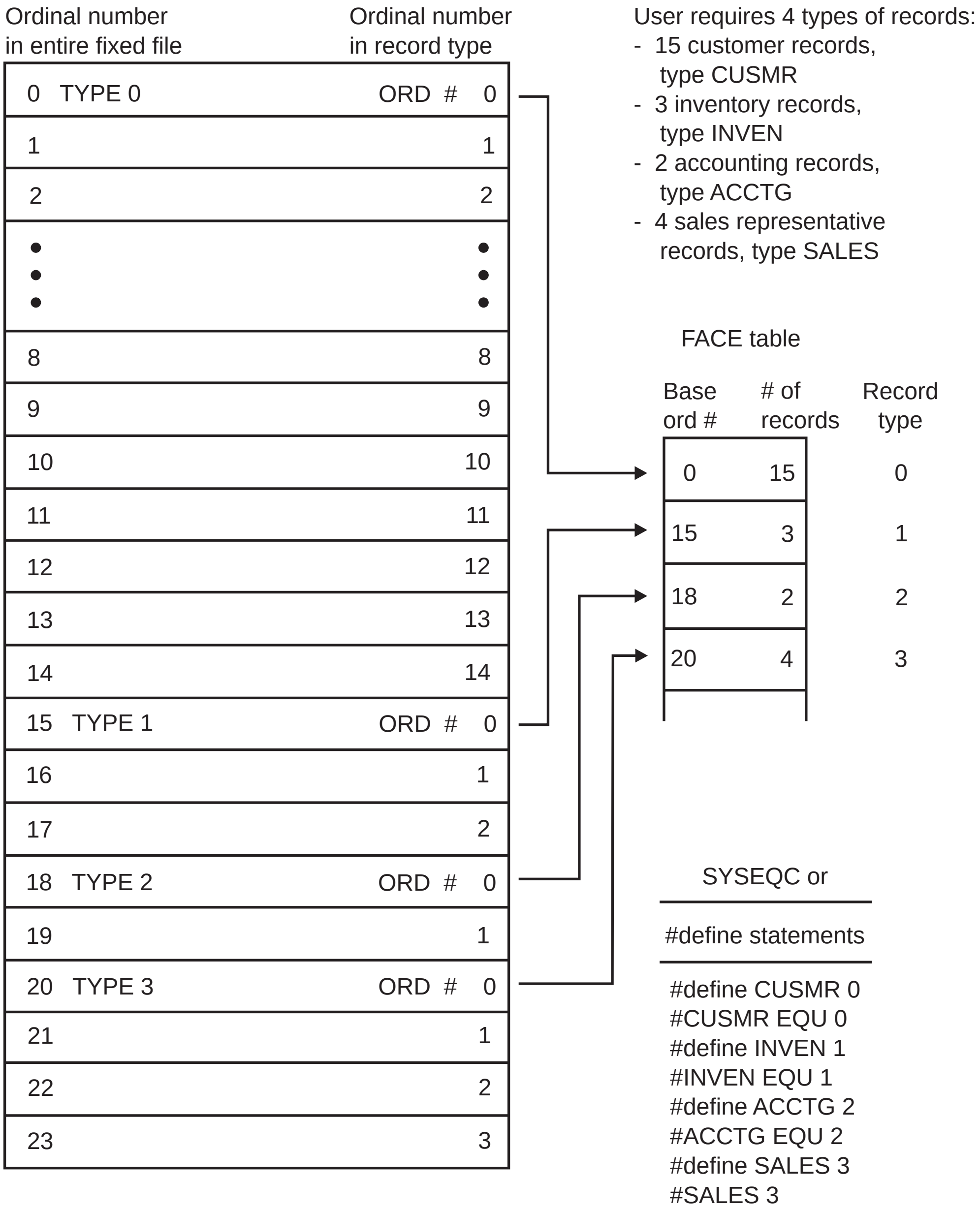


Figure 41. Example of Fixed File Organization and FACE/FACS Table

Application linkage from assembly language to FACE or FACS is by the ENTRC macro, by the `face_facs` function from ISO-C, and from TARGET(TPF) by the `#pragma linkage` and, optionally, the `#pragma map` preprocessor statements. For the FACS program or the FAC8C macro, the symbolic record type must be an 8-byte field, left-justified and padded with blanks, at the address specified.

Note: Entry requirements and return conditions for the FAC8C macro are different from FACE and FACS and are found through the IFAC8 DSECT in the IFACRET field. See *TPF General Macros* for more information about the FAC8C macro.

Register requirements on input for FACE or FACS, which in C language normally are set up by means of the `TPF_regs` structure, are as follows:

Register **Use**

Register 0 (R0)	The ordinal number in a record type.
Register 6 (R6)	The address of symbolic record type (for FACS) and record type (for FACE).
Register 7 (R7)	The location of an 8-byte field where the system file reference is to be placed. Normally, this location will be the FARW because the address must be there before the find or file function can be called.

Return conditions from FACE/FACS are as follows:

- Normal return
  - System file reference at the specified location
  - Register 0: maximum ordinal number for record type
  - Registers 1–5: contents unchanged
  - Registers 6 and 7: contents changed
- Error or Exception Return
  - Register 0: 0
  - Registers 1–5: unchanged
  - Register 6:

    when R7 = 1 and
    - R6 = 1, the requested record type is not in use.
    - R6 = 2, the record type is not defined or exceeds the limit.
    - R6 = 3, no records are defined for the record type on the associated SSU/Processor/I-stream.

    when R7 = 2 and
    - R6 = 0, the ordinal number exceeds the record type limit.
    - R6 not equal 0, the next valid ordinal, if requested, does not exist but does not exceed the maximum.
  - Register 7:

    = 1 - Input record type invalid
    = 2 - Input ordinal number out of range
    = 3 - No records defined for the record type on the requested SSU/Processor/I-stream.

    Note: If autostorage blocks are being used, then R7 is set to the address of the current autostorage block and the error code is placed in the autostorage block as the returned R7.

Return conditions from the FAC8C macro are found through the IFAC8 DSECT in the IFACRET field. See *TPF General Macros* for more information about the FAC8C macro.

Following is a FACS linkage coding example for assembly language:

```
L     R0, EBW020        Load ordinal number
LA    R6,=CL8'#VD1VD'   Symbolic record type
LA    R7,CE1FA2         Location = FARW, level 2
ENTRC FACS
LTR   R0,R0             Test for exception
BZ    EXCEPT            Logic for exception return
```

Making a Call to FACS Using TARGET(TPF)

The following TARGET(TPF) example makes a call to FACS to compute a file address prior to calling `findc`. Note that the assembler program FACS has been renamed for use in C as `getfileadd`.

```
#pragma linkage(getfileadd,TPF,N)  /* define appropriate linkage
#include <tpfapi.h>
#include <tpfio.h>
#define VD1RI "#VD1VD  "
struct TPF_regs regs;                 /* set up for register interface
 ⋮
regs.r6 = (long) VD1RI;               /* pointer to symbolic record ty
regs.r7 = (long) &(ecbptr()->ce1fa2); /* level where file addres
regs.r0 = 10;                         /* ordinal number                  *
&FACS.(&regs);                   /* calculate fixed file address    */
if (!regs.r0)                         /* FACS error?
  if (regs.r7 == 1)                   /* invalid record identification
    exit(0x12345);
  else
       {                              /* invalid ordinal number          */
           serrc_op(SERRC_EXIT,0x1234,"INVALID ORDINAL NUMBER",NULL);
       }

findc(D2);                            /* find the record                 *
```

`face_facs` Function

The `face_facs` ISO-C function provides for file address generation. The function is not available in TARGET(TPF). It initializes a file address reference word (FARW) with the data necessary to access a fixed file record.

The following generates a SON address for "`#PROG1`" record number 235 and stores it in the FARW for data level D6.

```
#include <tpfio.h>
⋮
unsigned long prog1_ordinal;
int rc = face_facs(235, "#PROG1", 0, D6, &prog1_ordinal);
switch (rc)
{
    case 0:  /* Success:  the FARW at data level D6 contains the    */
             /* file address for "#PROG1" record ordinal number     */
             /* 235, and prog1_ordinal contains the maximum record  */
             /* ordinal for #PROG1 fixed file records.              */

⋮
        break;
    Default: /* Error Conditions include the record type is not     */
             /* defined, is empty, contains fewer records than the  */
             /* number requested, and so forth.                     */


⋮
        break;
}
```

For information on the `face_facs` function, see the *TPF C/C++ Language Support User's Guide*.

Random Pool File Area

The dynamic requirements for file storage are maintained in an area called the *random pool file area*, or simply *pools*. Whereas the fixed file is allocated to static record types, the pools are allocated for random use and are dispensed on an as-needed basis. In other words, TPF manages the availability of each pool record. The application simply requests a pool record, uses it for as long as necessary, then returns it to the system.

Records in random pools are categorized by size (small, large, or 4 K), length of retention (short-term or long-term), and duplicate or single. Pools are also categorized by file address size (4-byte or 8-byte).

Short-term records are intended for quick turnover, such as the life of a customer transaction, and are returned immediately to the pool when the application releases them. Long-term records are intended for a longer interval of use dictated by application needs, and may extend into months. When released by application programs, long-term record addresses are written to tape for offline return to the pool.

A typical example of the practical use of the fixed file and the pools is provided by the airline reservation system. Flight inventory records and indexes to the passenger names are maintained on the fixed file. Pool records contain the individual passenger name records. Because the flight is flown on a scheduled basis, the inventory and indexes, though modified, will be retained permanently. When the flight has flown, however, individual passenger records are not retained online; they are placed in a history file and the pool space is made available for use again.

Duplicate records are records that, at allocation, are assigned to 2 different disk modules in order to increase availability of those records. If a module on which a duplicate record resides should fail, the record can still be retrieved from the other module. When the record is filed, both copies of the record are updated automatically. This updating is transparent to the application.

Application use of pool storage is simple and straightforward: it amounts to requesting an address (1 per function call) and releasing an address (1 per function call). Good use of resources mandates that addresses be returned promptly.

The assembly language GETFC macro gets small, large, or 4 KB records by pool ID. The pool ID is a parameter on the macro and is used as an index into the record ID attribute table (RIAT), which specifies long or short term, single or duplicate, and small, large, or 4 KB.

All file storage pool record addresses are returned by running the RELFC (release file storage) macro. The file address that is being returned is contained in the ECB data level or the DECB FARW specified in the macro instruction. See "Data Event Control Blocks" on page 29 for more information about DECBs.

Examples of these macros are as follows:

```
GETFC D6,,ID=AB         D6 specifies that the record address be
                        placed in the ECB level 6 FARW; AB is the
                        record ID, which is used in the RIAT.

RELFC D6                D6 specifies that the record address, in the
                        level 6 FARW, is to be returned.  This is the
                        only parameter required.  The record size and
                        whether it is short or long term is determined
                        from the contents of the FARW.
```

The C language `getfc` (obtain file pool address) function gets a file address (and optionally, a working storage block) based on attributes associated with a pool identification. The following arguments, listed in the order given below, are associated with the `getfc` function:

`level` or `decb`
`level` is a value of enumeration type `t_lvl` that specifies an available ECB data level. `decb` is a value of structure type `TPF_DECB*`, which is a pointer to a DECB.

`type`
One of the predefined terms `GETFCTYP0` through `GETFCTYP9`, which determine the size and pool types in the record identification attribute table (RIAT) associated with the `id` argument. `GETFCPRIME` and `GETFCOVERF` are still supported for migration purposes only.

`id` A pointer to a 2-character string that contains the pool identification. The pool identification is used as an index into the RIAT, which specifies long or short term, single or duplicate, and small, large, or 4 K.

`block`
One of the predefined terms `GETFCBLOCK` or `GETFCNOBLK`, which determines whether or not a working storage block with characteristics specified by the RIAT is to be obtained simultaneously.

`error`
One of the predefined terms `GETFCSERRC` or `GETFCNOSER`, which determines whether or not control is to be returned to TPF in the event of an error. `GETFCSERRC` will cause control to be transferred to the system error routine (with exit) in the event that storage cannot be obtained as requested.

The `getfc` function returns the requested file address.

All file storage pool record addresses are returned to the TPF system by calling the `relfc` (release file pool storage) function. The file address that is being returned is contained in the ECB data level or the DECB FARW specified as the only argument of the function.

Examples of Using the Pool Storage Functions:

It is important to understand that the pool area has no access scheme; that is, it has no record type, ordinal number, or other identification enabling retrieval except the system address. Once the application gets an address, it must maintain its own future access by saving that address. The address may be saved in the fixed file, the ECB, the global areas, or wherever good design dictates, but it is an application responsibility.

```
#include <tpfio.h>
#include <c$am0sg.h>
struct am0sg *amsg;                    /* Pointers to message blocks */
 ⋮
amsg = ecbptr()->ce1cr1;         /* Base prime AAA record      */
if (!(amsg->am0fch = getfc(D6,GETFCTYP3,"OM",GETFCBLOCK,
GETFCNOSER)))
                  /* D6 specifies that the record address be placed in the */
                  /* ECB level 6 FARW; pool type is 3; OM is the           */
                  /* record ID used in the RIAT; BLOCK causes a block and  */
                  /* file address to be obtained; NOSERRC does not cause   */
                  /* an exit to the system error routine in the case of an */
                  /* error.                                                */

   serrc_op(&EXIT.,0x33001," ",NULL,NULL);  /* Perform dump with exit     */
                                             /* if getfc() fails           */
 ⋮
relfc(D6);    /* D6 specifies that the record address, in the level 6  */
                  /* FARW, is to be returned. This is the only parameter   */
                  /* required. The record size and whether it is short- or */
                  /* long-term is determined from the contents of the FARW.*/
```

Figure 42. Using Pool Storage Functions

File Storage Access

Note: In this section, FARW refers to an ECB data level or a DECB. See “Data Event Control Blocks” on page 29 for more information about DECBs.

Whether fixed file or pools, all file accessing under TPF is done in a similar fashion. The TPF terms for file accessing are:

Term	Function Type
`find`	Read a record
`file`	Write a record.

There are many variations of these 2 basic operations, but the largest distinction exists between the “higher-level” and “basic” find and file functions. Higher-level functions allow an application programmer to read and write files without previously performing certain setup functions required for the basic functions. All basic find and file operations must be preceded by 2 application steps:

1. The system file reference (symbolic file address) must be supplied in the FARW.
2. The remainder of the FARW must be initialized according to the requirements of the function call used.

All higher-level functions must be supplied with the same information, but you must supply it in the argument list of the function itself.

Prerequisites for Basic Find and File Functions

The prerequisite information required by the basic find and file functions is as follows:

Step 1 The TPF system programs handle the details of obtaining the physical address of the record at the time of the find/file request. To enable this the application must supply the symbolic address in the low-order 4 bytes of the FARW. You can obtain this address in 1 of 3 ways:

1. If the record resides on the fixed file, pass the record type and ordinal number as a parameter to FACE/FACS. If the program specifies the FARW of a data level as the location, FACE/FACS will store the address there, ready for the find/file function request.
2. If a new pool record is to be used, use the `getfc` function to obtain the address and store it in the FARW.
3. If the address is a previously used pool record, move the address from where the application has saved it to the FARW. In C language, use the `ecbptr` function to do this. In assembly language, move it directly.

Step 2 As pointed out in the discussion of the ECB, the FARW is an 8-byte control field for file activity related to any data level. The low-order word must contain the file address. The high-order 3 bytes are used for data integrity checks, which vary as follows (the fourth byte is not used):

For *find* requests:

- The high-order 2 bytes can contain a record identification that, if present, must match the header of the record retrieved from file. If the record identification does not match, the ECB I/O error fields (`ce1sud` and `ce1sug` in DSECT EB0EB or header `c$eb0eb.h`) will indicate an identification check failure and control will return to the application.

  No check will be made if this field is zero in the FARW.
- The third byte, the record code check, can be used for a secondary integrity check; requirements are identical to the record identification.

For *file* requests:

- The high-order 2 bytes must contain a record identification that must match the first 2 characters in the header of the record to be filed. Otherwise, TPF will issue a system error and force the entry to exit.
- The third byte can contain a record code check (any nonzero value) to be used as a secondary integrity check with the same logic as the record identification.

  No check will be made if this byte is zero.

Use of the Higher-Level C Language Find and File Functions

The prerequisite information required by the basic find and file operations is supplied to the associated higher-level functions by means of the following arguments, which must be listed in the order given below:

`level` or `decb`

`level` is a value of enumeration type `t_lvl` that specifies available core block reference word (CBRW) and FARW locations. `decb` is a value of structure type `TPF_DECB*`, which is a pointer to a DECB.

`address`
A file address that is typically obtained through a call to either the FACE and FACS programs or the FAC8C macro.

`id` A 2-character string that must match the identification characters in the record image. For `find_record`, if the requested ID does not match the record's identification characters, the ECB error fields will indicate an identification check failure and control will return to the application at I/O completion. For `file_record`, if the requested ID does not match the record's identification characters, the system error routine will be activated and the ECB will be forced to exit. No comparison for either `find_record` or `file_record` will be made if the identification field is preinitialized to zero (X'0000') or is coded as `RECIDRESET`.

`rcc`
An unsigned character that must match the record control check (rcc) byte in the record to be retrieved (`find_record`) or filed (`file_record`). If the requested rcc does not match the record's rcc, then the system error routine will be activated and the ECB will be forced to exit. No comparison will be made if rcc is zero (X'00' or `'\0'`).

`type`
One of the predefined terms enumerated in `t_act` that are used to indicate the record's hold status.

The following optional parameters may also be used with find and file functions:

FIND_GDS
Use FIND_GDS to specify that the record to be read resides in a general file or general data set. If FIND_GDS is not specified, `findc_ext` accesses the record on the online database.

Note: If the flag is not needed default extended options flag, FIND_DEFEXT, should be coded. In this case user should consider using the `findc` function.

FILE_GDS
Use FILE_GDS to specify that the record to be filed resides in a general file or general data set. If FILE_GDS is not specified, `filec_ext` accesses the record on the online database.

FILE_NOTAG
The TPF system code that places the program identification in the record header is bypassed. This flag should only be used when the application updating the record has placed the required program identification in the header directly.

Note: If neither of the above flags are needed default extended options flag, FILE_DEFEXT, should be coded. In this case user should consider using the `filec` function.

Record Hold Facility

All programs should *hold* file records before updating them. The TPF record hold facility is intended to reserve a file record for exclusive use of 1 entry during record update; this ownership ensures data integrity and proper sequencing of updates.

TPF maintains a record hold table that consists of a list of file addresses and the ECB of the holding entry. When an entry attempts to hold a record by calling a find-and-hold function (`finhc`, `finwc`, or `find_record` with HOLD specified as the last argument), TPF honors the find request only if the address of the requested record is not in the record hold table. If it is, TPF denies the entry access to the record. ***This does not prevent entries that simply call a find function from gaining access to and even updating the record***. Only entries that properly use the hold facility are so restricted. Therefore, properly sequenced updates are only ensured if all programs cooperate in using the hold facility properly.

TPF's limitation of the hold feature to the individual record level is an important contribution to performance. Even so, misuse of the hold could have very serious effects on the system. A program should hold only 1 file record (or the unique first record of a chain of records) at a time; otherwise, a lockout condition may occur. (This is sometimes colloquially called *horns lock*, *deadlock*, or *deadly embrace*.) A program also should unhold a record as soon as possible—ideally, as soon as its update is complete.

Summary of File Reference Functions and Macros

Following is a summary of the find and file functions or macros, listed in alphabetic order. Basic and higher-level function types are so noted.

Table 22. File Reference Functions or Macros

Function or Macro	Summary	Type
`filec`	File a Record — Causes the TPF system programs to write a record to file. Both copies of the record will be updated if the record is duplicated. The application program supplies the file address, record identification, and record code check in the ECB data level FARW specified in the function call. The block of storage is returned to the appropriate storage pool following the I/O operation. No `waitc` should be used with this function; once `filec` is called, the application program cannot determine the status of the operation, and so cannot know whether I/O is complete.	Basic
`file_record` (C language only)	File a Record — Causes the TPF system programs to write a record to file. Both copies of the record are updated if the record is duplicated. The application program supplies the file address, record identification, and record code check in the function call, in addition to the data level of the CBRW that points to the record to be filed and the type of hold status to be associated with the record. One of 3 disposition types may be applied: **Type** **Disposition** `NOHOLD` The record written was not previously in hold status, and its associated block of storage will be returned to the appropriate storage pool following the I/O operation. `UNHOLD` The record's file address must have been held by the issuing ECB prior to the function call. Upon return, the record written will be removed from hold status, and its associated block of storage will be returned to the appropriate storage pool. `NOREL` The record written was not previously in hold status, and its associated working storage block will not be returned to the system. A programmer must code a `waitc` after a function call of this type ***only*** to determine the status of the file operation; the status of the operation cannot be determined for `NOHOLD` and `UNHOLD` calls, so `waitc` should not be used for these cases.	Higher-level

Table 22. File Reference Functions or Macros (continued)

Function or Macro	Summary	Type
`filnc`	File a Record with No Release — Causes TPF to write a record to file storage like `filec`, and requires the same application program input as `filec`. However, TPF does not return the block of storage containing the record to the storage pool, which leaves the block available to the application program after completion of the write. Because this function, unlike all other file functions, does not release the storage block, you must call `waitc` to ensure completion of the operation. If an error occurs during filing, TPF returns control to the system error routine.	Basic
`filuc`	File and Unhold a Record — Provides the same function as `filec`. However, the record is unheld before it is written to file storage. If another request to hold the record is pending, then a TPF system program effects a main-storage-to-main-storage move that saves a file access. (However, if the hold request is from another TPF system in a loosely coupled environment, then the record will be filed just as though there had been no hold request.)	Basic
`findc`	Find a File Record — Reads a record from file storage into main storage. The application program places the file address, and, optionally, the record identification and record code check, in the ECB data level FARW specified in the function instruction. A block of storage is obtained as a read-in area by TPF system programs and reference to the block is placed in the appropriate CBRW. You must call `waitc` to ensure completion of the I/O operation.	Basic
`find_record` (C language only)	Find a File Record — Reads a record from file storage into main storage. A block of storage is obtained as a read-in area by TPF system programs and reference to the block is placed in the appropriate CBRW. This function performs the equivalent of a `waitc` call, which ensures completion of the I/O operation. The application program supplies in the function argument the file address, and, optionally, the record identification and record code check, in addition to a CBRW and FARW data level for system use, and the intended hold status for the record. Either of 2 types of hold may be applied: **Type** **Disposition** `NOHOLD` The record address will not be placed in the record hold table following the I/O operation. `HOLD` The record address will be placed in the record hold table following the I/O operation. The application is responsible for ensuring that the record is removed from the record hold table prior to calling the `exit` function.	Higher-level
`finhc`	Find and Hold a File Record — Performs the same function as the `findc` function, in addition to including the record hold feature, which reserves a data record for the exclusive use of an entry during an update of the record.	Basic
`finwc`	Find a File Record and Wait — Performs the same function as the `findc` function while including an additional `waitc`. Thus, the application program need not call a separate `waitc` function to ensure completion of the I/O operation. TPF returns control to the system error routine on an error or abnormal I/O operation.	Basic
`fiwhc`	Find a File Record, Wait, and Hold — Includes all of the functionality of `finhc` with an additional `waitc`, as described for `finwc`, above.	Basic

Table 22. File Reference Functions or Macros (continued)

Function or Macro	Summary	Type
`unfrc`	Unhold a File Record — Causes the system service routines to unhold a record whose file address is contained in the ECB data level FARW specified in the function call. This function does not cause any records to be accessed.	Basic

Using Assembler Language File Reference Macros

In assembler language, all file reference macros require the ECB data level or DECB referencing the transfer as a parameter. In addition, the combination macros that include a WAITC must specify a label in the application program to which control will be transferred in any abnormal I/O operations. The application must then contain the logic to test the I/O indicators in the ECB (CE1SUG) to determine the nature of the abnormality and to specify the action to be taken.

Here are some macro coding examples:

```
FINDC D1          Find record, data level 1
WAITC ERROR1

FINWC D1,ERROR1   Find record, & wait, data level 1

FIWHC D2,ERROR2   Find record, & wait, hold file address,
                  data level 2

FILEC D6          File record, data level 6

FILNC D6          File record, do not release main storage
WAITC ERROR       block, data level 6

FILUC D6          File record, unhold file address, data
                  level 6

UNFRC D6          Unhold file address, data level 6
```

Note: Two specialized file retrieval macros—FNSPC (find a special record) and FDCTC (file data chain transfer)—do not examine the VFA area when searching for records. Therefore, when VFA delayed-filing is active, these 2 macros may retrieve records from file storage when more recently updated copies of the records exist in VFA. See the *TPF Database Reference* for more information about virtual file access.

Using C Language File Reference Functions

The following TARGET(TPF) example retrieves a data record from file on level `D1` or aborts if the retrieve is unsuccessful.

```
#include<tpfio.h>
 ⋮
findc(D1);
if (waitc())
   {
   serrc_op(&EXIT.,0x1234,"I/O ERROR OCCURRED",NULL);
       /*   serrc_op replaced the TARGET(TPF) exit: */
```

```
        /*   errno = 0x1234;                   */
        /*   perror("I/O ERROR OCCURRED");   */
        /*   abort();                          */
    }
```

The following example retrieves a data record from a general file onto level `D2`. The file address has already been computed and resides in the level `D2` FARW. Control is returned to the operational program when the I/O has completed and the record has been attached to the specified level.

```
#include<tpfio.h>
struct im0im *inm;
 ⋮
inm = finwc_ext(D2,FIND_GDS);
```

The following example retrieves a data record from file on level `D2` with hold. The file address has already been computed and resides in the level `D2` FARW. TPF returns control to the operational program when the I/O has completed and the record has been attached to the specified level.

```
#include<tpfio.h>
struct im0im *inm;
 ⋮
inm = fiwhc(D2);
```

The following example writes the data in the working storage block on level `D6` to a general data set, bypasses the record header update, and releases the block.

```
#include<tpfio.h>
 ⋮
filec_ext(D6,FILE_GDS|FILE_NOTAG);
```

The following TARGET(TPF) example writes the data in the working storage block on level `D6` to file or aborts if the retrieve is unsuccessful. The working storage block remains attached to the data level.

```
#include<tpfio.h>
 ⋮
filnc(D6);
if (waitc())
   {
   serrc_op(&EXIT.,0x1234,"I/O ERROR OCCURRED",NULL) ;
   }
```

The following example writes the data in the working storage block on level `D6` to file, bypasses the record header update, and releases the block. The file copy of the record is available to other ECB's on return.

```
#include<tpfio.h>
 ⋮
filuc_ext(D6,FILE_NOTAG)
```

The following example finds a message block on level 6 with hold and then removes the address from the record hold table.

```
#include<tpfio.h>
#include<c$am0sg.h>
struct am0sg *PRIME, *chain;      /* Pointers to message blocks    */
 ⋮
PRIME = ecbptr()->ce1cr1;     /* Base prime message block      */

/* Read first chain record with hold  */
chain = find_record(D6,(unsigned int *)&(PRIME->am0fch),
  "OM",'\0',HOLD);
unfrc(D6);                    /* Now remove the chain address from the */
                                  /* record hold table.                    */
```

Determining the Status of I/O Operations

The `waitc` function is used to delay processing of an entry until all I/O in process for that entry is complete. A single `waitc`, regardless of whether it is explicitly invoked or implied in another function call, applies to all pending I/O for the entry (it is not possible to restrict the `waitc` to 1 request). Control returns to the application program after all I/O requests are complete, and in C language the `waitc` function returns an integer value of zero.

In assembly language the WAITC requires a single parameter—the label of a routine in the requesting program—to which control will be transferred in case of hardware error or other abnormal condition. For example:

```
WAITC EXCEPT1
```

Should this routine be activated, the application must check I/O indicators in the ECB (CE1SUD, CE1SUG) to determine the condition and take appropriate action. (See “System Error Processing” on page 275 for more information).

In C language the `waitc` function returns the value of `ce1sug` in case of hardware error or other abnormal condition. The application must determine the condition and take appropriate action. (See “System Error Processing” on page 275 for more information).

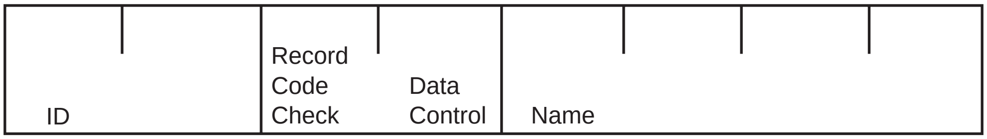
On many types of output operations, the data record is detached from the ECB upon execution of the library function call, in which case the `waitc` function is meaningless. In this situation, the application program cannot determine when output operations are completed. Standard system conventions must be relied on to handle errors.

The time for completion of an I/O operation is substantial compared with CPU processing speed. Accordingly, following the execution of `waitc` and during the delay pending I/O completion, TPF may transfer control to another entry. This may result in a transfer to the same program (or some other program) on behalf of a different entry. In addition to the explicit `waitc` function, many functions in TPF contain an implicit `waitc`. The *TPF C/C++ Language Support User’s Guide*, *TPF General Macros*, and *TPF System Macros* specify, for every C function and assembler macro, the circumstances under which a `waitc` is implied.

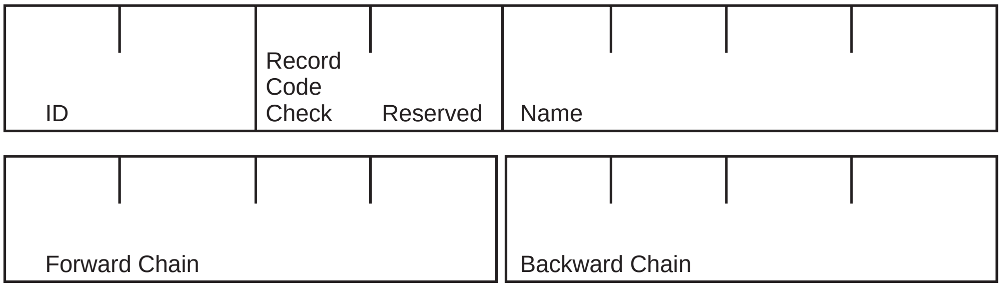
Standard Record Header

A standard header, which ranges from 8 to 32 bytes, is used for all data file records in the TPF system. The first 8 bytes comprise 4 fields that are present in all record types:

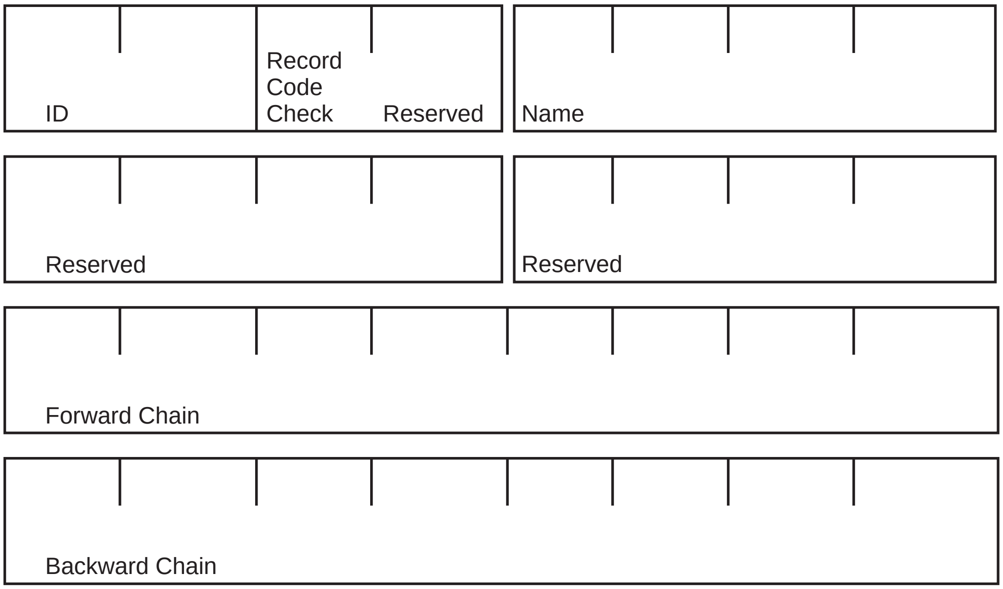
- Record identification (2 bytes)
- Record code check (1 byte)
- Data control (1 byte)
- Filing program ID (4 bytes).



The 8 bytes starting at X'008' of a standard header can contain either 4-byte forward and backward chain address fields or zeros. If these fields are not used for chaining, they can be used by the application for other data. The following shows a 4-byte file address standard header:



The 16 bytes starting at position X'010' of a standard header can contain either 8-byte forward and backward chain address fields or zeros. If these fields are not used for chaining, they can be used by the application for other data. The following shows an 8-byte file address standard header:

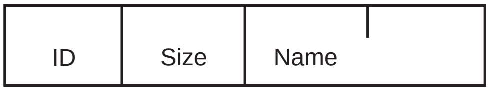


Program record headers have 8 bytes. The content of the header depends on the program attributes.

For the following items:
- File copy and core copy of file resident (FR) programs
- File copy of core resident (CR) programs

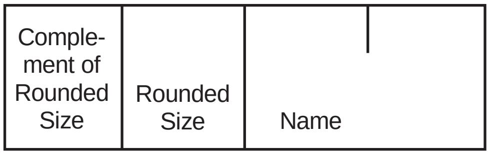
- Main storage copy of programs executing out of VFA or protected common blocks

the record header consists of the following:
- A 2-byte record ID
- A 2-byte compiled program size
- A 4-byte name.



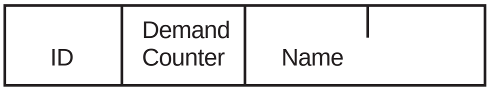
For the following item:
- Main storage copy of core resident (CR) programs

the record header consists of the following:
- The 2-byte complement of the core resident area reserved for the program
- The 2-byte size of the program's core resident area
- A 4-byte name.



For the following item:
- Unprotected common blocks

the record header consists of the following:
- A 2-byte record ID
- A 2-byte demand counter
- The 4-byte name.



Record Identification

This field is used by the system for a record integrity verification called identification or ID check. Assignment of record identifications (IDs) is a manual procedure involving a level of control to ensure that there is no duplication of identification assignment. Upon a request for I/O services, system action on the record identification varies as follows:

- For read requests, the high-order 2 bytes of the FARW can contain a record identification that, if present, must match the header of the record retrieved from file storage. If the record identification does not match, ECB I/O error fields will indicate an identification check failure and control will return to the application.

  If this field in the FARW is zero, no check will be made.
- For write requests, the high-order 2 bytes of the FARW must contain a record identification that matches the first 2 characters in the header of the record to be filed. Otherwise, TPF will issue a system error and force the entry to exit.

Record Code Check

An optional record code check may be requested by placing a nonzero value in the record code. This field is used by the application environment to do an additional data integrity check. This check may also be used as a security check. For example, the same code may be placed in all the records of a data chain that is only to be accessed by selected programs. The record code check procedures are identical to those used for the record identification check.

Data Control

This field is used to specify the record size and whether 2 standard chaining fields are to be used.

Program ID

Upon filing a record, the system inserts the 4-byte system name of the program requesting the file operation. For C programs, this is the name of the first external function. Use of the optional `NOTAG` parameter on file functions bypasses the TPF system code that places the program identification in the record header. This parameter should only be used when the application updating the record has placed the required program identification in the header directly.

Chaining Addresses

Standard forward and backward chaining address fields may be used to link records together. Complex data structures may, of course, be created by chains in the data content of the record. The use of the standard chain fields permits common system routines to do integrity checks on data chains. (This idea is used to recover unreturned or lost pool records, a process called RECOUP).

Tape Support

Real-time and general tape functions provide the application program with single file, multivolume tape input and output. In order to fulfill different system requirements, 2 distinct I/O capabilities are provided. The real-time tapes are intended for logging application data, system changes, system performance and other dynamic maintenance information in the real-time environment. The general tape functions provide the tape writing, reading, and searching capabilities required by utility and application programs.

Real-Time Operations

The real-time tapes are write-only tapes that are available to all entries in the system. All hardware-oriented tape functions such as labeling, label checking, and end-of-volume conditions are handled by TPF. The application program is relieved of these responsibilities and can assume that the real-time tapes are always available. Records are written on these tapes in the order in which the `toutc` and `tourc` functions are received by TPF. However, because each tape is also available to other entries in the system, the application program should not depend on creating consecutive records.

The `toutc` function writes a record, contained in main storage, on the specified real-time tape. This record must not be altered until the writing operation is complete. The execution of a wait system service routine ensures the completion of the I/O transfer. Note that no restrictions are placed on the record’s location in main storage. The record may be in a permanent main storage area, a working storage block, or the fixed work area in the ECB. To prevent a condition in which the tape output record is modified by another entry before the completion of the I/O transfer, it is recommended that the output record be located in a storage area held exclusively by the ECB. The following arguments must be specified in the order listed below when using the `toutc` function:

name
A pointer to a 3-character string that names the tape to be written to.

level
A value of enumeration type `t_lvl` that specifies the FARW containing the address of the record to be written to tape.

bufmode
One of the predefined terms `NOBUFF`, `NULL`, or `BUFFERED`, which determines whether the tape will be written to in immediate (`NOBUFF` or `NULL`) or buffered (`BUFFERED`) mode.

These parameters, which identify the symbolic tape to be used, are passed to TPF. The record starting location and byte count are stored in the specified request level in the FARW, in the following format:

Address of Start of Record	Unused	Byte Count
4 bytes	2 bytes	2 bytes

The CBRW is unused. Because of this, the application program may use any request level without releasing the main storage block held, if any, on that level.

The `tourc` function writes a record, contained in a main storage block, on the specified real-time tape. The main storage must be held by the ECB at the specified CBRW. This block is detached from the ECB upon execution of the function and then written to independently. Consequently, the application program cannot determine the status of the write operation; a `waitc` function call is unnecessary. Moreover, the request level used in the `tourc` function is available immediately for more use upon return from the function.

Code examples in assembly language:

```
TOUTC NAME=RTA,LEVEL=D2
TOURC NAME=RTL,LEVEL=D6
```

Code examples in C language:

```
  toutc("RTA",D2,BUFFERED);
  tourc("RTL",D6);
```

General Tape Operations

There may be active general tapes on the system at any time. These tapes are separated into distinct sets or groups, each one exclusively associated with a particular active entry: no entry can use another entry's general tape set concurrently. In addition to the tape reading, writing, and searching capabilities provided, the application program can refine and adjust the definition of its tape(s). Two types of C library functions are provided for general tape operations: basic and higher-level functions.

The basic general tape functions for assembly language and C language, which include `tasnc`, `tbspc`, `tdspc`, `topnc`, `tprdc`, `trewc`, `trsvc`, `tsync`, and `twrtc`, provide more direct control of general tape operations. In addition, the `tdspc_q` function is provided for C language only. These functions can provide very efficient general tape access when an entry can be assured of sole use of a general tape or when multiple I/O requests to a specific general tape are made. Each of the basic functions directly corresponds to a single macro service routine and therefore a small functionality, whereas the higher-level functions incorporate more system services. The judicious use of the basic functions can reduce unnecessary use of system resources and can be employed in some cases to produce more efficient code.

The higher-level general tape functions handle the details of general tape access for the application programmer, which can be especially useful in handling attempts to access a general tape that is already in use. The higher-level functions in C language include the `tape_open`, `tape_close`, `tape_read`, `tape_write`, and `tape_cntl` functions. It is important to note that these higher-level functions comprise an adequate library for performing general tape operations.

General Tape Functions

Most of the information for tape operations is described in terms of the basic tape functions. See the *TPF C/C++ Language Support User's Guide* and *TPF General Macros* for more information on basic and higher-level general tape functions.

Tape Allocation — Open and Close: The open and close functions (`topnc`, `tape_open`, `tclsc`, and `tape_close`) are used in determining the availability of a general tape to an application program.

A general tape first becomes available to an application program through the use of an open function. The following arguments must be specified when using the `topnc` function:

`name`
A pointer to a 3-character string that names the general tape to be opened.

`io` Indicates whether the tape is to be read or written to. `INPUT` indicates the tape is to be read. `OUTPUT` indicates that the tape is to be written to.

`bufmode`
The mode in which a buffered device will be written to. It can be one of the predefined terms `NOBUFF`, `NULL`, or `BUFFERED` `NOBUFF` or `NULL` indicates that a buffered device will be written to in write-immediate mode. `BUFFERED` indicates that a buffered device will be written to in buffered mode.

The `tape_open` function takes only the first 2 arguments listed above.

When an open function is called, the tape name is added to the tape set for that entry. TPF does the specified labeling operations and presents the desired tape positioned at the first data record. For `topnc`, TPF assigns the tape to the current ECB — making it unavailable to any other ECB. For `tape_open`, TPF places the tape into the reserved state — making it available for use by any ECB. Either close function, when called, deactivates the specified tape by deleting its tape name from the entry's tape set. TPF performs the trailer labeling when a close function addresses an output tape.

The allocation of physical tape units for general tape use is controlled by the computer operator, who is responsible for mounting the proper tapes and starting the various job phases in their correct sequence. The operator must know where new input or scratch tapes are needed, and when output tapes are complete. A utility program can communicate this information to the operator through the combined use of a setup sheet and the open and close functions.

Data Transfer — Read, Write, and Synchronize: The general tape input/output functions are `tprdc`, `twrtc`, and `tsync` (all basic functions) and `tape_read` and `tape_write` (higher-level functions). In addition to the I/O operation, each read and write function does the appropriate main storage allocation. During the execution of a read function, TPF gets a main storage block to read the record into. Reference to this block is placed in the specified CBRW when the input operation completes. Conversely, after the completion of a write function, the main storage block containing the record is released to the appropriate storage pool. Accordingly, each request of tape I/O via the general tape functions has an associated data level in the ECB; this CBRW contains the block references in standard format. The FARW at the associated data level is not used.

On return from the `tprdc` function, the status of the read operation is unknown. Consequently, a call to the `waitc` function must be made to ensure completion of the I/O in C language. The return from the `waitc` function is used to indicate the success or failure of any input operation. In assembly language the success or failure of any input operation is activated by the condition code. The `tape_read` function produces the equivalent of a call to `waitc`. Consequently, all pending I/O will have occurred prior to return to the calling program.

On a write function, the writing operation and main storage containing the data record are detached from the ECB on return from the function. Therefore, the data level used is immediately available for more use upon the return from the `twrtc` function. The same availability applies to the return from the `tape_write` function. Note that for both `twrtc` and `tape_write`, the status of the I/O operation cannot be determined by the application. Consequently, a `waitc` function call is irrelevant.

The `tsync` function can be called in conjunction with `twrtc` to ensure that all records contained in all buffers (the control unit hardware buffer and/or the host buffer) are physically written to tape. A `waitc` function call made afterward will ensure the operation completes. It also is desirable to call `waitc` before calling `tsync` to clear any outstanding DASD I/O. Note that the `tape_cntl` function using the predefined FLUSH command will perform a similar synchronization with the higher-level functions.

Tape Allocation — Assign and Reserve: It is important to note that the use of `tasnc` and `trsvc` is unnecessary when only higher-level functions are coded.

For those utility jobs that are single entry jobs (process from start to finish in the same entry), the previously described functions provide an adequate library. However, some jobs with extensive running time are segmented into a sequence of short phases. Each phase is a unique entry in the system and is begun by the computer operator upon completion of the previous phase. In such a case, it is desirable to have 1 set of tapes associated with the entire life of the job. An example of such a job might be nightly file maintenance, in which several capabilities are desirable:

- One input tape - processed in sequence by each phase of the job
- One output tape - processed in sequence by each phase of the job.

To eliminate the tape handling by the operator between job phases, the application programmer may use 2 special purpose functions, general tape reserve (`trsvc`) and assign (`tasnc`). These functions are designed to pass an open set of general tapes from a currently active entry, which is about to exit, to a future entry.

Between these phases the positioning of the tapes remains unchanged. The `trsvc` function reserves the specified tape, in the set of this entry or current job phase, for use by a future entry or the next phase. When the next phase of the job is initiated, the new entry makes the tapes available by calling the assign (`tasnc`) function.

Control Operations: Control functions provide the application program with the motion control necessary to search a tape. These functions include: `tbspc` (backspace), `trewc` (rewind), and `tape_cntl` (tape control). The `tape_cntl` function, depending upon its argument list, can be used to execute the following type of motion control:

- Forward space a specified number of physical blocks
- Backward space a specified number of physical blocks
- Synchronize a tape's buffer(s)
- Rewind to the first record of either the currently mounted tape or to the first volume of a multivolume tape set.

Not all applications can use tapes mounted in either blocked or unblocked mode. The following operational differences between blocked and unblocked tapes should be noted:

1. Forward space or backward space operations on a blocked tape are performed at the physical block level, rather than the logical record level.
2. If the `tbspc` function is issued to a blocked tape a system error will occur.

Tape Utilities: Tape utility functions provide information that helps applications manage tapes. The `tdspc` function returns a pointer to the data structure that contains tape status (`tpstat`). The `tdspc_q` function returns a pointer to the data structure that contains the module queue length (`tpqstat`) of a specified active tape.

Summary of General Tape Functions

Following is a summary of the general tape functions, listed in alphabetical order. Basic and higher-level function types are so noted.

Table 23. General Tape Functions

Function	**Summary**	**Type**
`tape_close`	Close General Tape — Deactivates a general tape from an entry (C only).	Higher-level
`tape_cntl`	Tape Control — Provides tape control operations not otherwise provided; for example, write tape marks (C only).	Higher-level
`tape_open`	Open General Tape — Makes a general tape available to an entry (C only).	Higher-level
`tape_read`	Read General Tape Record — Gets a main storage block and reads a general tape record (C only).	Higher-level
`tape_write`	Write General Tape Record — Writes a general tape record from a main storage block and releases the block (C only).	Higher-level
`tasnc`	Assign General Tape — Makes a general tape that was reserved by a previous entry available to the current entry.	Basic
`tbspc`	Backspace General Tape — Backspaces a general tape a specified number of physical blocks.	Basic
`tclsc`	Close General Tape — Deactivates a general tape from an entry.	Basic
TDCTC	The Tape Data Chain Transfer macro writes or reads a single record from or into specified main storage areas. WAITC must be issued to ensure that the CCWs have been completed. For a list of tape commands supported by this macro, see *TPF System Macros* (Assembly language only).	Basic

Table 23. General Tape Functions (continued)

Function	Summary	Type
`tdspc`	Display Tape Status — Provides the status of a specified tape.	Basic
`tdspc_q`	Display Tape Queue Status — Provides the module queue length of a specified active tape (C only).	Basic
TDTAC	The Tape Data Transfer macro performs 1 of several data transfer commands. For a list of tape commands supported by this macro, see *TPF System Macros* (Assembly language only).	Basic
`topnc`	Open General Tape — Makes a general tape available to an entry.	Basic
TPCNC	The Tape Control macro performs 1 of several control commands. For a list of tape commands supported by this macro, see *TPF System Macros* (Assembly language only).	Basic
`tprdc`	Read General Tape Record — Gets a main storage block and reads a general tape record. The tape remains open for use by other ECBs upon return.	Basic
`trewc`	Rewind General Tape — Positions a general tape to the beginning of its first data record.	Basic
`trsvc`	Reserve General Tape — Reserves a general tape at the current position for use by a future entry.	Basic
`tsync`	The Synchronize Tape function enables an application to ensure that records contained in a buffer are physically written to the tape.	Basic
`twrtc`	Write General Tape Record — Writes a general tape record from a main storage block and releases the block.	Basic

Operator Control of Tape Operations

The section discusses a method of structuring general tape programs to allow for independence from the hardware tape configuration. The application programmer provides the computer operator with a run sheet outlining the symbolic tape configuration for each job. This run sheet identifies each tape by general name and attributes (input, output, scratch, blocked, and so on). Using the run sheet, tape units are assigned and the necessary tapes are mounted for the job. The operator then communicates the tape units assigned and their corresponding general tape names to TPF.

Each tape is recorded as being in a ready status by TPF. When the utility job is started, the application program opens each tape with a call to either `topnc` or `tape_open`. TPF expects to find a ready tape for each `topnc` or `tape_open` call from the application program. If no such tape exists, TPF sends a message to the operator requesting the desired tape. Return from the open function is delayed until the tape is made ready.

Only the operator and TPF are concerned with the tape units in use. Thus, the application program, working with symbolic general tape names, is independent of the system’s physical tape configuration. For details about C function use or macros, see the *TPF C/C++ Language Support User’s Guide*, *TPF General Macros*, and *TPF System Macros*.

General Data Set and General File Support

General data sets are usually related to some offline processing. Either data is produced online (for example, management reports) to be processed by offline programs, or data is produced offline (for example, programs to be loaded) for online processing. A general data set provides a data interface between the offline and online system components.

A TPF general data set (a disk module) is related to the meaning of an MVS data set. The records of a data set are allocated sequentially in the same module (called a volume in MVS). The file is processed online by using the TPF find and file functions. The find and file function extensions have a parameter where GDS can be specified to indicate that a requested data record resides on a general file or general data set, rather than in the TPF online database. A special assembler macro, file data chain transfer (FDCTC), permits the processing of records that are not restricted to the standard TPF record sizes. A file with standard record sizes is processed online by using the TPF find and file functions. The file referencing is slightly different from the procedures used to access fixed record types and pool records.

TPF provides 2 distinct general data set functions to allow an ECB-controlled program to access records in a TPF general data set:

Function	**Action**
`gdsnc`	General Data Set Name — Associates a data set name with a unique entry by using the MVS data definition name (DDNAME) concept. (This is functionally similar to an MVS OPEN.)
`gdsrc`	General Data Set Record — Accesses a specific record in the data set named by the `gdsnc` function. (This corresponds to the use of FACE/FACS for obtaining a file address).

A command allows an operator to mount and dismount modules (volumes) associated with a data set and to make an association between a data set name and a data definition name. The combination of `gdsrc` and any find function is equivalent to an MVS GET.

A TPF general file is a sequentially organized set of data that, in principle, is similar to a general data set. However, general files are not MVS compatible. Limited classes of general files are used by system programs. They may be used by application programs if appropriate.

General files and data sets are exceptions to the online real-time environment and will not be discussed here in more detail. For more information about general data sets, see *TPF Database Reference*.

General Data Set Functions

The general data set name (`gdsnc`) and general data set record (`gdsrc`) functions are used to access general data set records. `gdsnc` is used to open and close the data set. `gdsrc` is used to access specific records in the data set. The find and file functions are used to read from and write to the data set.

The general data set (GDS) support uses the MVS concept of data definition name (DDNAME) specified by the program and data set name (DSNAME) bound by the operator at execution time. The program, when coded, specifies a 16-character DDNAME as a parameter for the `gdsnc` function. The operator, when mounting the general data set for the program, specifies both the DDNAME and the DSNAME as part of the mount command. Then, when the `gdsnc` function is issued, the TPF system provides the binding to link the program to the correct data set as specified by the operator.

Note: Nothing prevents the DDNAME and the DSNAME from being the same.

The user must open the data set with the `gdsnc` function by passing the DDNAME, volume sequence number, and relative record number of the data set. The volume sequence number and relative record number may be 0. `gdsnc` returns in the CE1FMx, CE1FCx, CE1FHx, and CE1FRx fields of the file address reference word (FARW) that the user specified with the function, the indexes necessary for the system to access the data set. The file address of the relative record in the data set is returned in CCHR format in the file address reference word extension (CE1FXx). If the relative record number was 0, the address of the first record of the data set is returned.

To access specific records in the data set, the user must pass with the `gdsrc` function the relative record number of the desired record. The relative record number may be zero. `gdsrc` returns the file address of the desired record in CCHR format in the file address reference word extension (CE1FXx). If the relative record specified was 0, the address of the first record of the data set is returned.

The find or file functions with the GDS parameter specified are used to read data from or write data to the general data set record.

After manipulating the data with the find and file functions, the user must close the general data set with the `gdsnc` function. The parameters passed are the same as those passed with `gdsnc` open. The FARW (CE1FMx, CE1FCx, CE1FHx, CE1FRx) is cleared on return from `gdsnc` close.

The following is an example of the use of the general data set macros for assembly language.

```
***********************************************************************
* "OPEN" THE GENERAL DATA SET.                                        *
***********************************************************************
         XC    CE1FM8(1),CE1FM8      VOLUME SEQUENCE NUMBER
         MVC   EBW000(16),=CL16'TPF.DATA.SET.A'  DDNAME
         MVC   EBW016(4),=F'0'       RELATIVE RECORD NUMBER
         LA    R14,EBW000            ADDRESS OF DDNAME
         GDSNC D8,O,RCT=A,SIZE=L,WORK=YES  OPEN DATA SET
         LTR   R14,R14               CHECK RETURN CODE
         BNZ   GDSNERR               BRANCH IF ERROR
***********************************************************************
* READ, UPDATE, AND WRITE THE FIFTH RECORD OF THE GENERAL DATA SET.   *
* WHEN WORK=YES THE USER MUST PASS THE RELATIVE RECORD NUMBER IN THE  *
* FOUR BYTES IMMEDIATELY FOLLOWING THE DDNAME.                        *
***********************************************************************
         LA    R14,EBW000            ADDRESS OF DDNAME
         MVC   EBW016(4),=F'5'       RELATIVE RECORD NUMBER = 5
         GDSRC D8,SIZE=L,WORK=YES    GET ADDRESS OF 5TH RELATIVE RECORD
         LTR   R14,R14               CHECK RETURN CODE
         BNZ   GDSRERR               BRANCH IF ERROR
         MVC   CE1FA8(2),=CL2'GS'    RECORD ID
         MVC   CE1FA8+2(1),=XL1'00'  RECORD CODE CHECK
         FINDC D8,GDS=Y              READ DATA SET RECORD
         WAITC FINDERR               BRANCH IF ERROR
           .
           .                         UPDATE THE DATA SET RECORD
           .
         FILEC D8,GDS=Y              WRITE DATA SET RECORD
         WAITC FILEERR               BRANCH IF ERROR
***********************************************************************
* READ, UPDATE, AND WRITE THE THIRD RECORD OF THE SAME DATA SET.      *
* WHEN WORK=NO THE USER MUST PASS THE RELATIVE RECORD NUMBER IN THE   *
* FARW EXTENSION.                                                     *
***********************************************************************
         MVC   CE1FX8(4),=F'3'       RELATIVE RECORD NUMBER = 3
         GDSRC D8,SIZE=L,WORK=NO     GET ADDRESS OF 3RD RELATIVE RECORD
         LTR   R14,R14               CHECK RETURN CODE
         BNZ   GDSRERR               BRANCH IF ERROR
         MVC   CE1FA8(2),=CL2'GS'    RECORD ID
         MVC   CE1FA8+2(1),=XL1'00'  RECORD CODE CHECK
         FINHC D8,GDS=Y              READ DATA SET RECORD
         WAITC FINDERR               BRANCH IF ERROR
           .
           .                         UPDATE THE DATA SET RECORD
           .
         FILUC D8,GDS=Y              WRITE DATA SET RECORD
         WAITC FILEERR               BRANCH IF ERROR
***********************************************************************
* "CLOSE" THE GENERAL DATA SET.                                       *
***********************************************************************
         LA    R14,EBW000            ADDRESS OF DDNAME
         XC    CE1FM8(1),CE1FM8      VOLUME SEQUENCE NUMBER
         MVC   EBW016(4),=F'0'       RELATIVE RECORD NUMBER
         GDSNC D8,C,RCT=A,SIZE=L,WORK=YES  CLOSE DATA SET
         LTR   R14,R14               CHECK RETURN CODE
         BNZ   GDSNERR               BRANCH IF ERROR
```

Input Device Support

A package exists to provide common input device support. You can use this package when doing sequential 4 KB reads from 1 of the following devices: tape, general data set, virtual reader, or a user-defined medium. Once a data definition name has been set up for the device, you can use that name to perform operations. Use the ZDSMG DEF command to define the DD name. The input support package takes the data definition name as input and performs an open, read, or close on the device. The application does not need device-specific information; only the data definition name is needed. See *TPF System Installation Support Reference* for more information.

System Error Processing

Errors may occur at any point of processing. There may be programming errors, such as incorrect function parameters, hardware malfunctions, and a variety of unusual conditions such as identification check failures on file operations. Three levels of errors are:

1. Hardware malfunctions that are overcome by retrying the I/O operation. In this case, error statistics are recorded, but the entry is insulated from the problem (for example, unit checks).
2. An error is detected by TPF from which the programs related to an entry may be able to recover. In this case the ECB-controlled program regains control.
3. An error is detected by TPF or the hardware from which the entry cannot recover (for example, an addressing exception generated by an ECB-controlled program). In this case the entry is forced to exit.
4. An error is detected that makes any more processing inadvisable. This is called *catastrophic failure* (for example, an operation exception in TPF). Such a failure is detected by various components of TPF and may require a system restart.

The general philosophy of TPF system error processing is:

- Save as much information as possible for the technical staff's analysis and action. The amount of data saved is dependent on the severity of the error and its nature, whether program error or data integrity, for example.
- Notify the central site that the error occurred. A message to the CRAS terminal will identify the error by number and the program in control at the time of the error. All system errors are documented in *TPF Messages, Volume 1* and *TPF Messages, Volume 2*, which describe the cause of the error, the action taken by TPF, and in some cases suggests actions to be taken by the application program or operations staff.
- Respond to the program in control at the time of the error, if possible.
- Determine if more processing may continue.

If an ECB-controlled program is given control after an error is detected, then error indicators are set in the affected ECB. The program is assumed to contain the procedures to respond accordingly. These procedures typically include:

- Providing a *selective* main storage dump of data pertinent to the active ECB
- Sending an error message to the CRAS terminal
- Either regaining control or forcing an exit of the ECB.

An important reason for returning control to the application (when possible) is to allow response to online terminal operators. The application can respond to the operator in the most meaningful manner once the condition is analyzed. If TPF is unable to return to the application program, it will force a standard message to be sent to the operator/terminal: “Check data and call supervisor”.

Given the many possible errors and variations in the nature and complexity of applications, it is not meaningful to generalize about what the application should do. Designers and applications programmers must be alert to the possibility of errors at any point, and diligent in making the best possible response based on what is known. Additional information about TPF system error processing can be found in *TPF General Macros* and *TPF System Macros*. Error processing with the control program save area in the ECB is discussed briefly in “Control Program Save Area” on page 38.

SYSRA Macro

The SYSRA macro enables assembly language application programs to:

- Determine the action to be taken.
- Specify the error number.
- Specify the error number prefix.
- Append additional error message text to the error number when it appears on the CRAS console.
- Branch to a symbolic location in the event of a hardware error.
- Branch to a symbolic location if the control program detects an invalid file address.

For a complete description of SYSRA, see *TPF General Macros*.

Standard Error Functions

The TPF system provides several C functions that can be used for standard error processing by an application program. They are:

- `serrc_op`

- `serrc_op_ext`
- `snapc`

All of these functions can be used to generate a main storage dump. The `snapc`, `serrc_op`, and `serrc_op_ext` functions determine whether, after dump processing, the ECB is to be exited or the control returned to the application program. It is an application's responsibility to test the I/O indicators in the ECB (`ce1sug`) to determine the nature of the abnormality and specify the action to be taken.

The `serrc_op` function enables an application to:

- Force the ECB to exit, or not
- Associate a specific system error identification number with the generated dump
- Send a specified message to the CRAS console
- Display additional main storage areas on the generated dump.

In addition to the `serrc_op` capabilities, the `serrc_op_ext` function enables an application to associate a particular prefix with a system error identification number to determine which user application generated the dump or whether it was an IBM program that generated the dump.

The `snapc` function allows an application to:

- Determine the action to be taken
- Determine the error number
- Include the registers in a snapshot dump to the system console
- Specify the program name that is issuing the `snapc` function
- Specify the error message text to be appended to the snapshot dump
- Specify if the subsystem (SS), subsystem user (SSU) and terminal address should be taken from the ECB
- Specify the location of the `snapc_list struct` which indicates the locations and lengths of the areas to be dumped.
- Indicate the prefix to be put on the identification code for the snapshot dump.

Using C Language Error Functions

Following are several examples of C language error functions:

1. `serrc_op` function

   The following example generates a main storage dump bearing the identification number U012345 (U is the default prefix). "ERROR OCCURRED" is the message displayed at the prime CRAS and appended to the dump. TPF returns control to the application program after generating the dump.

   ```
   #include <tpfapi.h>
    ⋮
   serrc_op(&RETURN;,0x12345,"ERROR OCCURRED",NULL);
   ```

2. `serrc_op_ext`

   The following example generates a main storage dump bearing identification number A012345 (A is the user-chosen prefix). "ERROR OCCURRED" is the message displayed at the prime CRAS and appended to the dump. TPF returns control to the application program after generating the dump.

   ```
   #include <tpfapi.h>
    ⋮
   serrc_op_ext(&RETURN.,0x12345,"ERROR OCCURRED",'A',NULL);
   ```

3. `snapc` function

This example forces a snapshot dump bearing ID number 12345 with a prefix of 'A' to be issued. Control returns to the program after the dump. The registers are included in the dump. The ECB is used for the SS and SSU names, and terminal ID and the program name are "C001". The `snapc_list` is `snapstuff` and the message is "PROGRAM BLEW UP".

```
#include <tpfeq.h>
#include <tpfapi.h>

test()
{
 struct snapc_list *snapstuff[2]

 snapstuff[0]->snapc_len = 4;
 snapstuff[0]->snapc_name = "MYSTUFF ";
 snapstuff[0]->snapc_tag = ecbptr()->ebw000;
 snapstuff[0]->snapc_indir = SNAPC_INDIR;

 snapstuff[1]->snapc_len = 0;

 snapc(SNAPC_RETURN,0x12345,"PROGRAM BLEW UP",snapstuff,\
       'A',SNAPC_REGS, SNAPC_ECB, "C001");

 exit(0):
}
```

Temporarily Detaching and Attaching Main Storage Blocks

The DETAC macro or the `detac` or `detac_ext` (detach) functions allow you to temporarily detach the main storage block on a given ECB data level or DECB without releasing it. Use this facility if:

- You need a main storage block but no more ECB data levels are available.
- You need a main storage block but you are not sure which ECB data levels are available when your program gains control from another program.
- You need to keep using a particular DECB, but it is not available.

You must reattach the block with the same data level in that ECB using the ATTAC macro or the `attac` or `attac_ext` function.

Excessive use of the `detac` and `attac` functions can cause a depletion of working storage: you should carefully monitor their use.

Design Considerations

There are some design considerations that the application programmer should be aware of. These issues overlap, to some extent, the responsibilities of the application designer.

Program Sharing in Main Storage

TPF allows all active entries to share the programs that are currently in main storage. Whenever an external function is called, TPF first determines if its program segment is already in main storage or previously requested by another entry, in which case a new retrieval is not required. It is this feature that makes it essential that all programs be reentrant. Any number of entries may be using the same program segment, but each at a different point in the logic flow.

Program sharing is a key of TPF's high performance. However, you must assume the potential for a file retrieval every time a program is entered. Processing for every message type should be designed to minimize the requirement.

Virtual File Access Facility

You may have some data that is highly accessed, but only temporarily or intermittently. In this case, you would like to have fast access (requiring no I/O) to the data when it is needed, but you are reluctant to allocate the data to main storage permanently.

The virtual file access (VFA) facility temporarily retains both program and record data that has been retrieved from file storage in a specially allocated area of main storage. A file reference request searches the VFA area for the data to be accessed; if it is not found, then the search moves to file storage, but if it is found, the relatively slow I/O operation to file storage is avoided. Note that if the data is found in the VFA area, then any wait system service request to await the completion of I/O does not reset the 500-millisecond timeout counter.

Data remains in the VFA area until the VFA area is full; then the least used data is filed and removed from the VFA area to make room for more frequently used data. In this way, data that is occasionally highly accessed (such as records that are updated every hour) remains in main storage during high activity, but can be filed to file storage when activity slows.

Program Organization

ISO-C programs do not have the 4KB size limit that TARGET(TPF) programs do. If the need for organizing programs into packages comes mainly from this size constraint, performance gains can be realized by using calls to functions that reside in a single ISO-C program module. The performance gains result from avoiding the Enter protocol usually used for calling programs.

Typically, TPF programs are organized into packages. A package is a group of program segments designed to handle the processing for a given functional area. There is no rule about the number of segments in a package or the breadth of functionality covered. This is a function of the complexity of the application. The functionality to be programmed must be assessed and then broken into packages. The principal criteria should be efficient program development and, ultimately, good system performance.

An example of possible package structure is offered in “Application Message Editor” on page 44. All the functions of the application message editor might be grouped into a package. But again, it is a question of application complexity and variety of message types. Large applications will find it advisable to have multiple packages just to handle input. For example, some airlines have implemented a function relating to reservations, for fare quotation and ticket printing. The input messages may be long and involved. A separate package was designed just to edit input for a fare quote/ticketing request and pass the reformatted request to the quotation and ticketing functional packages.

Another possible approach is to group all programs required to handle a specific transaction into a package. For example, in the banking environment, the creation of a new customer account would be a transaction consisting of multiple messages, each supplying a data item required for the new account. Similar transaction types, those requiring essentially the same type of data and accessing the same files, might be combined into a single package.

It is outside the scope of this document, which is intended primarily as a programming guide, to offer a definitive discussion of application design. Following is a survey of some of the principal design considerations, and some miscellaneous programming tips.

Modular Programming

Modular programming is the concept of dividing the problem solution into its logical parts or routines so that each part may be programmed independently. This should contribute to ease of understanding, ease of modification and standardization of structure. Ideally, any module could be modified without affecting other modules. The structured programming concept of top down logic should be used as much as is practical without sacrificing performance. The major decision criteria and the main path logic should be developed first.

Performance Considerations

TPF application and system programmers must remain aware of the need to make their programs perform efficiently. Evaluate every routine against this objective. Some compromises are to be considered, such as coding productivity, simplicity, ease of understanding and modifying—but performance must always be a primary factor.

File Access

The most productive area for improving performance is in minimizing file accesses. The main focus for this, of course, is the data structure itself and the method by which the program accesses its data. Other factors, however, are directly controlled by program design and are therefore in the hands of the individual programmer.

- Develop conventions for procedures for using records shared by multiple programs, including the ECB data levels or data event control blocks (DECBs)to be used. When possible, these records should be kept in main storage and passed via the ECB from program to program—not filed by program A, then retrieved by program B. It was suggested in an earlier section that it is important to release system resources, such as main storage blocks, as early as possible.

However, that determination must be made in terms of the total processing for the entry, not just in the logic of each program segment.

In a multiple processor (I-stream) environment, programmers must also be aware of shared main storage resources and should minimize their use. When required to access a shared resource, appropriate multiprocessing techniques must be used. See *TPF Concepts and Structures* for additional discussion of multiprocessing.

Early development of conventions on data level use also will allow all programs to be coded with specific records on the conventional level, thus avoiding excessive use of the `attac` and `detac` functions.

- Do not file records without determining that they, in fact, have been updated. Set an indicator when you update the record, and file only if the indicator is set.
- Do not file partial updates of a record. Complete all updates, then file the record once.
- Hold file records only for as long as required for a complete update: holding file records can be a significant performance bottleneck. If the update is complete but the entry still requires the data block, file the block without releasing it (`filnc`), and then unhold it (`unfrc`). The data block may then be retained as long as necessary without serious system impact.

Note: You can use DECBs for passing data between programs without using ECB data levels. For more information, see “Data Event Control Blocks” on page 29 .

Coding Techniques

Other performance-oriented suggestions:

- For areas where fast performance is most critical, consider using low level C functions, if they are available. For a 1-time use it may not seem significant, but consistent use of the most efficient instructions can be productive.
- Evaluate moving and clearing operations. Do not clear work blocks unless necessary, and then choose efficient methods.
- Evaluate different table search techniques; under some conditions a binary search can be dramatically faster than a serial search.

Ease of Modification and Expansion

It should be assumed that all application programs will require modification at some point. Ease of understanding and modification is therefore of primary importance. Programs should be designed and coded so that external changes such as system configuration and values or other program changes have the least possible impact. The key to this is to keep coding symbolic. Absolute values should be avoided; use length attributes and calculations from symbolic references. All assembler programs and TARGET(TPF) functions are limited to 4KB and ISO-C functions have no size limit in main storage. Assembly language and TARGET(TPF) programs should leave room for expansion (perhaps 20% of record size) when they are originally coded.

Program Commentary

Emphasize good programming commentary. It serves the dual purpose of explaining how the program works and communicating expected interfaces with other programs.

As suggested earlier, application designers should develop conventions about the use of common system resources, such as data levels, the ECB work area, and shared interface records such as the routing control block and the scratchpad area. Whether or not by convention, program commentary must specify expected interfaces: the expected condition of common system resources at the time the program is activated and when it passes control to other programs. The program listing should contain clear, concise comments throughout. When modifying existing code, ensure that comments remain meaningful.

Utility Segments and Subroutines

Special care should be given to developing routines that will be common to many programs. These programs will probably be frequently accessed and should therefore be coded for maximum efficiency. The effort might be assigned to experienced programmers and the code assigned permanently to main storage. All programmers should be aware of the availability of these utilities and ensure optimum use of them.

Miscellaneous Programming Tips

- Audit loop control for efficiency and brevity. This can improve performance and reduce object code size.
- Aim at efficient code without over sophistication. There is virtue in simplicity: its ease of understanding and modification, and less probability of errors.
- Generate responses that are meaningful to both programmers and general user personnel. Consider a message table from which programs may request formatted responses by number.
- Be aware of each function’s return conditions. Consult the function specifications.
- Call any system error function at the point of error so the dump will show all conditions at the time the error occurred. Do not do any clean-up procedures, or pass control to another program prior to calling a system error function.
- Do not request dumps for information only. No processing can be done during a dump; inputs are queued. Performance will be degraded by improper requests for dumps.

  In addition, do not request dumps with too high a severity. Prefer using `snapc` instead of `serrc_op`. Do not dump more information than you really need. Use the LISTC function to specify the important areas in main storage for a dump.
- Use create functions with discretion. Creating a new entry requires some overhead. More importantly, if entries are created indiscriminately without monitoring, system resources such as main storage blocks could be depleted.

TPF Testing Environment for Assembly Language

A key element in the TPF assembly language application environment is a comprehensive set of testing facilities. In a real-time system that must be relied on by online users 24 hours a day, no program can be incorporated into the online system until it has been thoroughly evaluated for correct logic, effective accomplishment of its prescribed function, and the ability to interface with TPF system programs and other application programs.

Test System Characteristics

The TPF test facilities are designed to provide:

- A check on violations (or possible violations) of programming conventions
- An orderly progression from simple debugging through complex multiprogram tests, including the entry of messages from terminals
- A uniform data definition and database for use in all levels of testing
- The ability to batch various test runs
- A flexible method to specify and modify data for each test case
- A method of simulating unavailable programs
- Flexibility in specifying the types of output desired
- Online components to assist in the detection of faulty programs
- Offline components to print the results of a test
- Debugging aids (traces, formatted dumps, processing snapshots).

Testing Levels

TPF test facilities are designed to be used at the following levels:

- Package or transaction testing:

  This refers to the testing of several programs together to check the validity of interrelated functions within a package of programs. It may include testing a complete transaction in a single-thread or multithread environment. *Multithread* means that concurrent entries must be processed by the package. Thus, the reentrant programming conventions are verified.
- System testing:

  This is a multithread test through a realistic simulation of the environment in which the programs will ultimately operate.

Test System Components

The following components of the test facilities will be discussed in somewhat more detail in this section:

- System test compiler (STC):

  This is used to create a test database, input test messages, and control information for a test unit. STC runs offline under control of MVS and is the primary vehicle for test and online data preparation.
- Program test vehicle (PTV):

  PTV creates a testing environment that runs under TPF. When PTV is activated, it provides comprehensive checks on application programs by controlling the execution of test cases. During PTV testing, live terminals can be active simultaneously. PTV can only be executed in a uniprocessing environment, meaning only one instruction stream can be active.

- Real-time trace (RTT):

  This is used to monitor and record the activity of application programs in the TPF system. RTT can be run either in an operational system or when testing the control program macro activity. The level of output detail is controlled by option indicators in commands when used by an operational environment, and by control statements when used in a PTV environment.
- Selective file dump and trace (SFDT) and diagnostic output formatter (DOF):

  The SFDT writes specified file records to tape and, in conjunction with DOF, formats all file traces and printouts of main storage for ease of analysis.

System Test Compiler (STC)

All testing requires test data. Three types of data are generally required: program data, data representing system or application records, and message data. The system test compiler (STC) programs provide the basic tool for this data preparation. Figure 43 on page 285 shows the STC environment. STC is also used to generate initial system and application data records, including global area data.

STC, executing offline under MVS control, generates a test unit tape that is the primary input for testing under the program test vehicle (PTV). The test unit tape contains one or more test units. A test unit is generated via STC statements in the form of card image inputs. Some of the input is merely copied to tape and represents commands ultimately interpreted by PTV or DOF. Other input contains statements that invoke STC offline programs to insert programs and data into the test unit. The programs and data in an STC-generated test unit are used by PTV to modify a TPF set of online files with a function equivalent to a TPF system load. When PTV is in control, the set of online files is ordinarily called a test database. The following card image inputs are processed by STC to form a test unit.

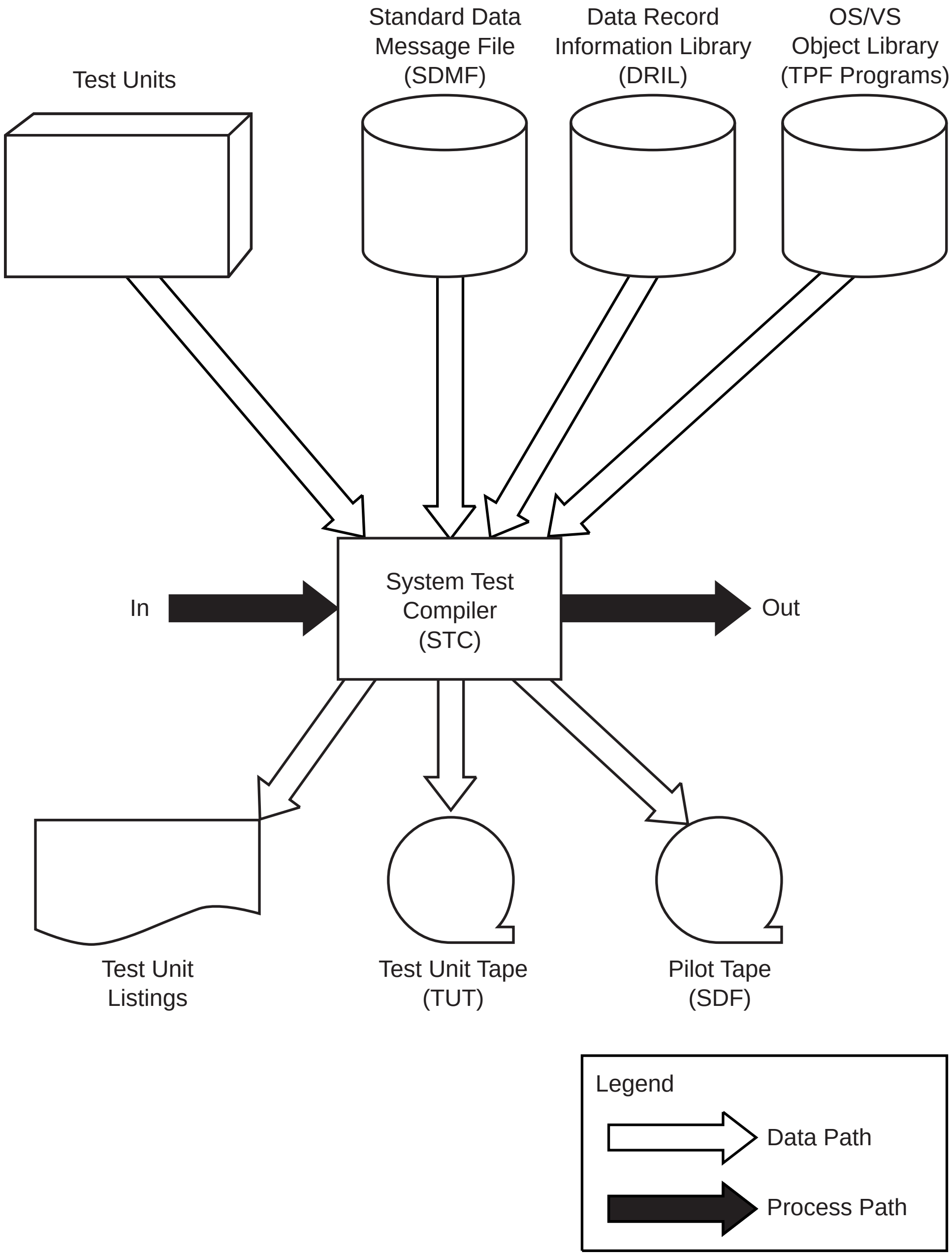

Test Units
Standard Data Message File (SDMF)
Data Record Information Library (DRIL)
OS/VS Object Library (TPF Programs)
In
System Test Compiler (STC)
Out
Test Unit Listings
Test Unit Tape (TUT)
Pilot Tape (SDF)
Legend
Data Path
Process Path

Figure 43. System Test Compiler

RUNID	This identifies the beginning of a test unit. The card image is placed on tape for use by PTV.
PTV options	These options are used by PTV and RTT to control test options, and will be described in the following sections. STC places an exact image of the following PTV option card images on the test unit tape: • Terminal simulation: Specifies printer output to be formatted like terminal output. • Dump options: Specifies test data to be collected.
Data	This is used by STC to generate a sentinel on the test unit tape which marks the beginning of data records for PTV.

Data Card Images (STC Instructions)

Data records and test messages are generated with the same procedures. STC either extracts records or messages from a file called Standard Data/Message File (SDMF) or by generating them through the use of data declarations placed on a file called Data Record Information Library (DRIL). The DRIL file contains a data declaration of all the records in the TPF system for which there is a corresponding declarative macro in the TPF macro library. The DRIL declaration of a record is composed of a series of assembly language statements which define the data record. The STC user is, therefore, capable of generating data to be placed on a test unit by using STC statements and the macro name and definition of the record. A data record or message is produced with an STC generation start (GSTAR) instruction.

For example:

```
AM0SG     GSTAR 1
AM0LIT    ENT X'020406'.
```

causes STC to create an input message and place the value 020406 in the terminal address field of the message. (Absolute values rather than symbolic references can be used to create data records and messages).

The SDMF file contains “canned” data records and messages. The same procedure used to generate data records and messages through a DRIL reference is also used to generate the canned data contained on the SDMF file. This data can be transferred to the test unit tape by using the appropriate STC segments.

All data records generated by STC contain a 200-byte prefix that is used to communicate address information to the loading program. The address information usually consists of record types and ordinal numbers which are converted, via FACE or FACS, to absolute file addresses at load time. These card images are illustrated in Figure 44 on page 287.

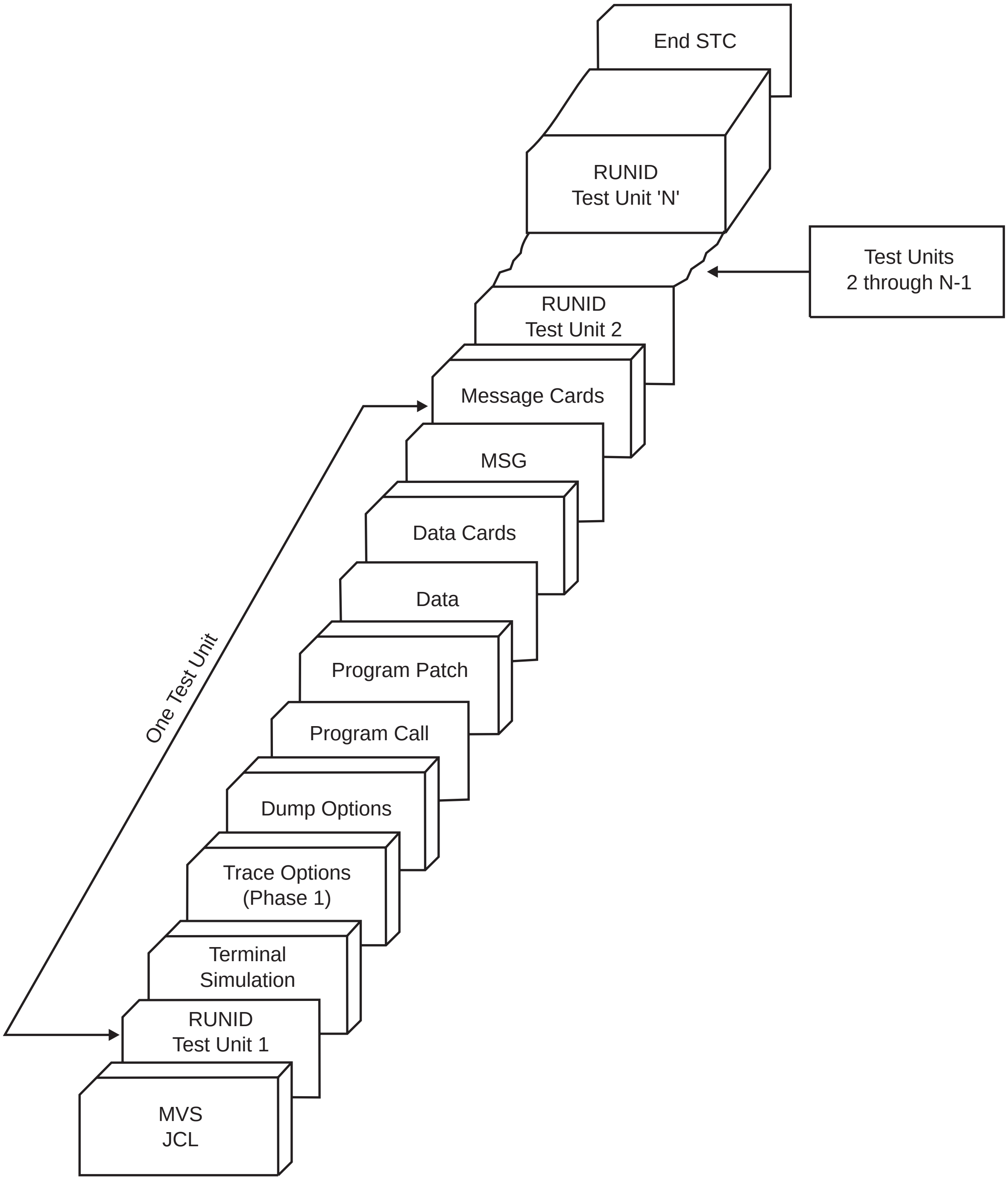


Figure 44. Multiple Test Units

MSG This is used by STC generate a sentinel on the test unit tape which marks the beginning of messages for PTV.

Message Card Images
These are the procedures used to create messages are identical to those used for data records.

In addition to generating the test unit tape input for PTV, STC may also be used to generate pilot tapes. These tapes contain predefined sets of data records which can be loaded onto the TPF files either by the TPF system loader facility or by PTV. This facility can be used either in a test environment or to load initial data for an operational environment.

For further details on the use of STC see the following documents:
- *TPF Operations*
- *TPF Program Development Support Reference*.

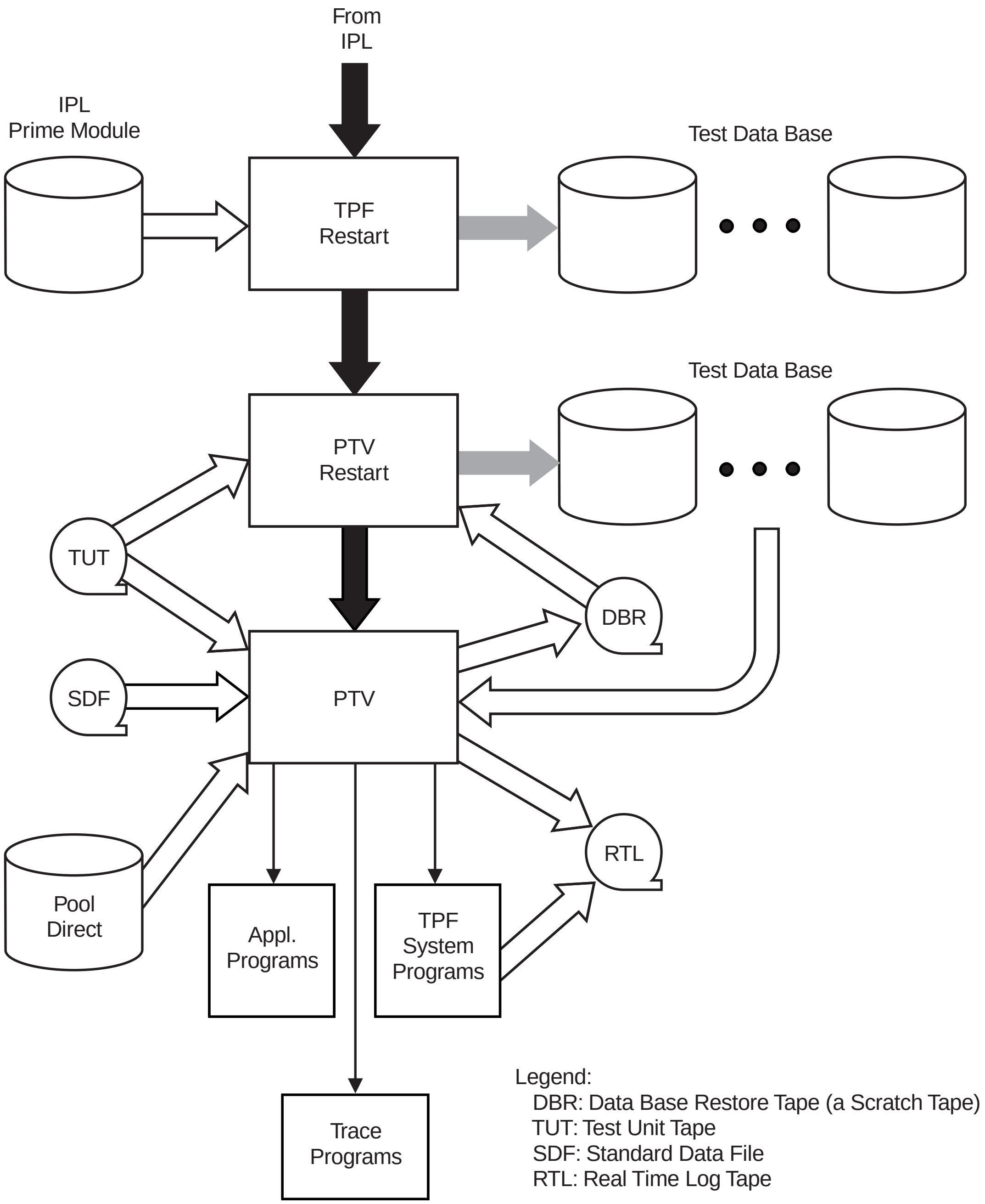
Program Test Vehicle (PTV)

PTV controls the execution of test cases. In order to execute PTV in a multi-I-stream environment, the system must be IPLed in uniprocessor mode since the PTV facility is not capable of multiprocessing. PTV testing requires the use of the TPF system loader to create the equivalent of online files.

Files adhering to the standard TPF data structure but generated for the purpose of testing are the test database and are maintained using the TPF system loading facility. Data can be dynamically and temporarily replaced or added by the PTV loading facilities for the duration of a test.

Following the IPL sequence and TPF system initialization, the TPF restart scheduler program enters the first PTV program which starts the test environment.

PTV provides extensive test facilities to execute application programs in an environment which runs under the control of the TPF system control program. Inputs to PTV are one or more test units on a test unit tape (TUT) created by the system test compiler (STC). STC may also be used to create an input pilot tape (SDF) for PTV which contains application data records. A file pool directory file is used to initialize the pool directories. Information generated by a test run is written to the real-time log tape, RTL, or the real-time tape, RTA, for later processing by the diagnostic output formatter (DOF). Figure 45 on page 289 represents an overview of PTV.



Legend:
DBR: Data Base Restore Tape (a Scratch Tape)
TUT: Test Unit Tape
SDF: Standard Data File
RTL: Real Time Log Tape

Figure 45. Program Test Vehicle

The following capabilities are provided by PTV:

- Builds the file environment from data provided in a test unit
- Restores the database before each test unit is executed, if so specified, and at the end of all testing.
- Provides message input in single-thread or multithread mode of operation
- Permits live message input in all phases of testing
- Monitors and records activity of application programs
- Records TPF macro activity and input/output messages
- Records file updates.

Control information generated by STC and placed on a test unit tape (TUT) is used to specify PTV options for each test unit in unit and package testing. A system restart is used to invoke PTV. Before PTV processes an input tape, an operator is requested to input a command to identify the type of test, that is, unit/package test or system test. If the test is a system test, then some of the PTV options must be selected by a command, otherwise the options are selected with parameters in the RUNID statement. The significant PTV options are:

- Request to load pilot tapes and/or pool directories: It is specified in RUNID.
- Message input mode—burst or asynchronous: *Burst mode* means that if there are no active entries, then a specified number of messages from a test unit are placed on the input list. *Asynchronous mode* means that a specified number of messages are kept in the system until the messages in the test unit are exhausted. The mode is specified in RUNID.
- Live input may be permitted between any test unit of any phase; it is specified in RUNID.
- PTV dump options require separate card image input. These options permit the specification of the amount of information to be dumped and permit the selection of conditions upon which dumping should occur. Macros as well as macro groups are used to identify dumping conditions. Examples are:
  - Dump the ECB on all enter/back macro requests.
  - Dump the ECB and attached data blocks on all FIND macro requests.

  A real-time trace (RTT) option which corresponds to a PTV dump option must be specified for dumping to take place. PTV dump options select additional information for output. RTT options are specified with commands in the message stream.
- Terminal simulation options

  These options are used to request terminal simulation by the Diagnostic Output Formatter on messages whose addresses are specified on separate option cards.
- Database Restore options

  A highly useful function of PTV is the database restore option. PTV will save every online record modified by the test run, using the database restore tape (DBR), and will either:
  - Restore those records to their original status after each test unit, or
  - Build on the processing of each test unit, and restore the records at the end of the test run.

  An indicator in the RUNID card specifies the option desired.

Package Test

Package test may be at the individual program level or the package level. It depends on standard TPF facilities; no special test macros are provided (or allowed). It is ideal for testing interfaces among system and application programs in a controlled environment.

Package testing consists of:

- Data is loaded from the test unit tape.
- PTV options are selected from control information generated by STC and placed on the test unit tape. (The online trace options RTT and SFDT are permitted in package testing but must be requested with a command in the message stream on the test unit tape).
- Test data is restored by PTV at the end of a test.

- An ECB is created by TPF as the result of input message processing (the input messages are generated by STC and placed on the test unit tape).

System Test

The system test environment consists of a major application or class of applications loaded on the test database files and a TPF system which includes the PTV programs activated. Generally the files include a structured test database which eases the analysis of test results. The only input placed on the test unit tape is a collection of input messages (no PTV options, programs or data). PTV options are selected at the beginning of a test run with a command. The application is driven by creating multiple entries from the messages found on the test unit tape. The application is, in essence, in an online environment. Terminals may be used in addition to the test unit tape input.

Live Test

At times, the test database is used without PTV programs activated. Terminals are used for input instead. This means that test SCRIPTs are provided to the people who enter messages. This is often called live testing. Real-time trace (RTT) options and selective file dump and trace (SFDT) options are selected through the use of commands. PTV options may not be used.

For additional information on the use of PTV see *TPF Operations* and *TPF Program Development Support Reference*.

Real-Time Trace (RTT)

The real-time trace program (RTT) is used to monitor and record the activity of application programs in the TPF system. RTT can be run in an operational system or when testing under the control of PTV. The output of the RTT is a historical record of input message and control program macro activity. The level of detail of the output is controlled by option indicators in commands when used in an operational environment and by control cards when used in a PTV environment.

RTT sets indicators in the ECB to indicate which macros are to be traced and the output required. Options may be selected to trace:
- Specified macros in all entries
- Specified macros in specified programs
- File type macros referencing specified file records.

Macros and macro groups (for example, all Find macros) are used to select conditions upon which RTT should provide trace information. The type and amount of data included in the trace option may be specified to include only a simple logging of the macro execution or more extensive logging. The logging option may be further modified by the PTV dump option if RTT is used in conjunction with PTV package testing. All output is written to the real time log tape.

For additional information, see the *TPF Program Development Support Reference*.

Selective File Dump and Trace (SFDT)

The selective file dump and trace program (SFDT) provides a debugging aid for file type activities. Like real-time trace, selective file dump and trace can be run on the online system or when testing under the control of PTV. All output is written to the real time log tape.

The selective file dump programs write specified file records (primary and chain if desired) to the real-time log tape. Functional messages during online operation and control cards during PTV testing may be included at any point in the processing to help determine the point at which records are altered.

The selective file trace option dumps only updated records from a set of specified file records during a file trace period. At the end of the period, only the updated records are written to the real-time log tape. If any record is updated more than once, the intermediate updates are not recorded and only the final contents are output. This function enables information to be obtained about specific file record update activity for a given period of time. A subfunction provides a list of addresses of all file records updated during a trace period. Online command or PTV control cards define the trace period.

For additional information, see the *TPF Program Development Support Reference*.

Diagnostic Output Formatter (DOF)

The program test vehicle (PTV), real-time trace (RTT), selective file dump and trace (SFDT), TPF system error dump routine and CCP I/O trace routines produce output which is written to the real-time log tape (RTL). The Diagnostic Output Formatter (DOF) deblocks, decodes, and formats the real-time log tape data in the form of an easily readable printer listing used for debugging. DOF is executed offline under MVS control.

System test terminal simulator (STTS) programs are operated in conjunction with DOF. Each of the simulators formats and prints input and output messages as the message would appear on a specified terminal. The printed output from each of the simulators is used as a debugging aid to check message format and content.

The real-time log tape which becomes input to DOF may be processed at any time after a test run. DOF is made up of the various subroutines that format the logical record data types found on the input tape. Encoded data is formatted and interpreted for ease of readability. For example, the SVC interrupt code in the old PSW is printed as a macro mnemonic as well as a hexadecimal value. Labels are assigned to significant main storage. The ECB work area is in hexadecimal with the EBCDIC translation (if possible) on the line below. Data blocks attached to a unique ECB are grouped with that ECB for output.

For more information, see the diagnostic output formatter in *TPF Program Development Support Reference* and *TPF Operations*.

The programming event recording (PER) facility enables applications programmers to monitor specific events in a native TPF 4.1 environment. Use the ZSPER command to trace:

- Storage alteration events
- Instruction fetching events
- Successful branching events (on ESA/390 systems only).

The information you supplied in the data parameter of the ZSPER command is compared to the particular event you are monitoring in the range you specified. If a match is detected, a PER interrupt occurs. Using the ZSPEP command specify an output device with the PRINTER parameter. This parameter lets you specify a device that is meaningful to your installation. The default is the RO CRAS. You must write a program to support the information provided at the PER exit. See *TPF Program Development Support Reference* and *TPF Operations* for additional information.

Debugging Programs and Diagnosing Problems in C Language

This chapter is designed to help you determine where or why a problem is occurring.

Do not use the diagnosis, modification, or tuning information as a programming interface.

Run-Time Debugging

Errors that occur while running a program can be vastly more difficult to correct than syntactic errors. This section contains some topics that will help.

Function Mismatches

This is one type of error to check when you are having a problem.

Description

When a program calls a function, the stub associated with the function provides an index into the AOLA and an index into the corresponding library vector. Online this identifies the executable code for the function. If either of these indexes do not match the online system, unexpected function calls can result.

There are several places where this can go wrong.

- Stub information is a product of the (offline) library build tool. The stubs indicate the order that libraries and functions must be loaded online. Offline, the order is specified by the input to the library interface tool.
- If the order that 2 libraries are loaded online is switched, the order of the LIBVECs is also switched. This is analogous to being in a building in an office on the floor directly below the office where you want to be.
- If the order of functions loaded online is switched from how they are defined by the library build tool, the wrong function is executed.

Refer to the IDSLST DSECT for the structure of the AOLA.

Indications

To determine whether this problem exists compare the name of the function being called with the name of the code being executed.

The name of the function code being executed is found at the beginning of the executable code identified by the LIBVEC. Refer to the library build script for the library name and LIBVEC index for the function. Assume for the following example the function is FUNC in library CTAL.

The library ordinal number for a given library can be displayed online by doing the following:

1. Lock the library load module into main memory using `ZRPGM`. The address returned is the starting address of the loadset.

```
ZRPGM CTAL LOCK
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
RPGM0001I 19.56.44 PROGRAM CTAL LOADSET BASE LOCKED IN CORE
------------------ AT ADDRESS 00820BB0
```

2. Display the loadset entry for the library load module using `ZDCOR`. This provides the library name and address of its library vector.

```
ZDCOR 820BB0.10
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 19.56.44 BEGIN DISPLAY
 00820BB0- 0000FFFF C3E3C1D3 00822D28 00000000 ....CTAL ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

3. Using `ZDCOR` display main storage at the address shown by the previous step. The value 00000001 is the ordinal number of CTAL.

```
ZDCOR 822D28.10
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 19.56.44 BEGIN DISPLAY
 00822D28- 00000001 00820BB0 00823E60 00823EB8 ........ ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

4. Using `ZDCOR` display main storage at the function address displayed by the previous step. Assume the function has LIBVEC ordinal 2. This puts it at 00823EB8.

```
ZDCOR 823EB8.10
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 19.56.44 BEGIN DISPLAY
 00823EB8- 4700C004 C6E4D5C3 18781894 5850C006 ....FUNC ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

Identifying the Library Ordinal Number

A library ordinal number can be found using the shared library names table (SLNT).

Description

As the library addresses are filled into the AOLA structure, the 4-character name of the library load module is filled into the SLNT. The library ordinal number is used to place the library name into the SLNT.

Indications

The SLNT is located in the CCISOC CSECT and can be displayed online by doing the following:

1. Display the address of the CP link map using `ZDDCA LMP`

```
ZDDCA LMP
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DDCA0001I 19.56.44 DUMP TAG LMP  ADDRESS - 000B9BB0
```

2. Display the CP link map using `ZDCOR` and find the address of CCISOC.

```
ZDCOR B9BB0.40
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 19.56.44 BEGIN DISPLAY
 000B9BB0- C3C3D3C9 C4C8F4F0 000BB000 00000500 CCLIDH40 ........
 000B9BC0- C3C3C9E2 D6C3F4F0 000BC000 00003850 CCISOC40 ........
 000B9BD0- C3C3D3C1 D5C7F4F0 000C0000 00009500 CCLANG40 ........
 000B9BE0- C3C3D4E3 C8F1F4F0 000C9500 00000002 CCMTH140 ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

3. Display CCISOC using `ZDCOR`. The SLNT is located at the beginning of CCISOC. Libraries CISO and CTAL are ordinal numbers 0 and 1.

```
ZDCOR BC000.100
CSMP0097I 19.56.44 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 19.56.44 BEGIN DISPLAY
 000BC000- 000BF034 000BC01C 000BD024 00000000 ..0..... ........
 000BC010- 00000000 E2D3D5E3 00000000 C3C9E2D6 ....SLNT ....CISO
 000BC020- C3E3C1D3 40404040 40404040 40404040 CTAL
 000BC030- D8D7D5F0 40404040 40404040 40404040 QPN0
 000BC040- 40404040 40404040 40404040 40404040

⋮
 000BC0E0- 40404040 40404040 40404040 40404040
 000BC0F0- 40404040 40404040 40404040 40404040
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

ISO-C Dynamic Load Modules (DLMs)

The TPF DLM startup code (CSTRTD) provides a bridge between TPF Enter/Back services and the ISO-C environment. Because CSTRTD receives control whenever an ISO-C DLM is called, CSTRTD must be defined as the first object in the load module. The entry point function will be called from within CSTRTD and, when the function ends, control returns to CSTRTD. Because the entry point is resolved in the TPF startup code, the prelinker and linkage editor list the following message:

```
IEW2650I 5102 MODULE ENTRY NOT PROVIDED.  ENTRY DEFAULTS TO SECTION CSTRTD
```

CSTRTD obtains the address of the entry point function through the weak external @@DLMENT. @@DLMENT is resolved by the loader to the address of the function with a name equivalent to the name of the load module.

For example, load module DLM1 has an entry point function named DLM1. If there is no function called DLM1, the TPF offline loader returns with a condition code 8 and lists the following reason:

```
DLM ENTRY PT NOT FOUND IN MODULE.
```

Stub routines are used to resolve the VCONs produced by the compiler for library function calls and external function calls, which can reside in BAL segments, TARGET(TPF) segments, or other ISO-C segments. It is important to inspect the output of the prelinker to verify the correct stubs are merged into the load module.

@@DLMENT is used to get the address of the DLM prolog entry point. The address of @@DLMENT itself comes from the link map generated during linkage editing.

To display online the address of @@DLMENT, do the following:

1. Lock the C load module into main storage using the `ZRPGM` command with the `LOCK` parameter:

   ```
   ZRPGM CIM9 LOCK
   CSMP0097I 15.45.12 CPU-B SS-BSS  SSU-HPN  IS-01
   RPGM0001I 15.45.12 PROGRAM CIM9 LOADSET BASE LOCKED IN CORE
   ------------------ AT ADDRESS 01F91690
   ```

2. Use the `ZDCOR` command with the address shown in the previous step. @@DLMENT is at offset X'14' from this address. In the following example, the DLM entry point is at address X'1F91700'.

```
ZDCOR 1F91690.20
CSMP0097I 15.45.12 CPU-B SS-BSS  SSU-HPN  IS-01
DCOR0010I 15.45.12 BEGIN DISPLAY
 01F91690- 0000FFFF C3C9D4F9 00000000 01F91988 ....CIM9 .....9..
 01F916A0- 00000000 01F91700 00000000 00000000 .....9.. ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED
```

The following example of the file map shows the two object files specified in the prelink job and all the stub routines automatically included by the prelinker.

```
========================================================================
|                               File Map                              |
========================================================================

*ORIGIN  FILE ID  FILE NAME

   PI     00001   DD:OBJLIB(CSTRTDNA)
   PI     00002   DD:OBJLIB(CIMANC)
   A      00003   ISO0000.DEVP.STUB.OB(@CINIT)
   A      00004   ISO0000.DEVP.STUB.OB(STRPBRK)
   A      00005   ISO0000.DEVP.STUB.OB(CRFA)
   A      00006   ISO0000.DEVP.STUB.OB(EXIT)
   A      00007   ISO0000.DEVP.STUB.OB(CIMZ)
   A      00008   ISO0000.DEVP.STUB.OB(ATOI)
   A      00009   ISO0000.DEVP.STUB.OB(CIMC)
   A      00010   ISO0000.DEVP.STUB.OB(CIMB)

*ORIGIN:  P=primary input     PI=primary INCLUDE    SI=secondary INCLUDE
          A=automatic call     R=RENAME card         L=C Library
```

Storage for ISO-C Static Variables

If the compiler determines that 1 or more static variables are declared in a given source file, the object file needs to be processed by the prelinker. The linkage editor returns a condition code 8 and lists

```
@@XINIT@ as an UNRESOLVED EXTERNAL REFERENCE
```

for object files that contain static data but have not been processed by the prelinker. @@XINIT@ is defined to be the address of the static data length and is filled in by the prelinker.

Every time an ECB enters a DLM or a library that contains static, the startup code tests @@XINIT@ to see if there is an associated static area. If the load module contains static data, @@XINIT@ contains the address of the static data length; otherwise, it contains zero. The static data pointer in the TCA is saved in the startup code's stack frame. The static exception routine is called to find the current load module's static data area. The pointer that is returned by the static exception routine is saved in the TCA.

After the entry point function ends, the startup code restores the saved static data pointer to the TCA before returning to the control program.

ISO-C static frames are mapped by the IDSCSF data macro.

Layout of ISO-C Structures in a Dump

System error processing formats the contents of the ISO-C control structures, the ISO-C language support stack, and all static areas if the ISO-C environment exists.

The ISO-C TCA appears in the dump following the program area. The dump tags identify the following:

TCA	Beginning of the TCA structure
BOS	Address of the beginning of the ISO-C stack
EOS	Address of the end of the ISO-C stack
WSP	Address of the current writable static area
EPT	Address of the entry point function
CID	Beginning of the CID structure
WSC	Beginning of the writable static control block (WSCB)
CTC	Beginning of the CTCA structure
TPS	Beginning of the TPSA structure.

```
*ISOC CONTROL AREA

01000010  TCA 00000000 00000000       BOS 01001964 01009FFF EOS       00000000 00000000           00000000 00000000
01000030      00000000 00000000           00000000 00000000           0100A08C 0071C930           01009FFC 80A17E04
01000050      0071C930 0700C198           01000200 00000000           00A17AB0 00300000           800F35CA 01002FFF
01000070      01000010 01009F6C           00A17DF0 000BF7C0           00000000 000BF7C0           00000000 98000000
01000090      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01000130      00000000 00000000           00000000 00000000           00000000 00000090           01000010 00002FF0
01000150      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
010001B0      00000000 00000000           0700C198 0700C198           0700C198 0700C198           0700C198 0700C198
010001D0      0700C198 0700C198           0700C198 0700C198           0700C198 0700C198           0700C198 0700C198
010001F0      0700C198 0700C198           0700C198 0700C198           01000A64 00000000 WSP       4E000000 00000000
01000210  PRM 00000000 00A17C7C           00030B02 010011F4           000BD024 01001544       EPT 00000000 00000004
01000230      00000000 00000000           00000000 00000000           00000000 00000000           01000250 00000000
01000250  CID 000BF7C0 000BF584           000F32C8 000F36A8           01000264 00000000 WSC       00000000 00000000
01000270      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01000A50      00000000 00000000           00000000 00000000           00000000 00000620 CTC       00000000 00000000
01000A70      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01000B70      00000000 0084134E           00004650 00842240           00841E48 00841F00           00841EB0 00841F88
01000B90      00841F68 00842178           00842088 0084F040           00842198 00000000           00000000 00000000
01000BB0      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01000C50      00000000 00000000           00000000 00000000           0000E2E8 E2C9D540           404000E2 E8E2D7D9
                       .
                       .
                       .
                       .
010018D0      00000000 00000000 TPS       00000000 00000000           00000000 00000000           00000000 00000000
01001950      00000000 00000000           00000000 00000000           00000000
```

Figure 46. ISO-C Control Area Dump

The data in the ISO-C stack is broken down into the individual stack frames and each is tagged with the name of the function that created it.

The initial stack frame will always be the stack frame associated with TPF startup code CSTRTD when the ISO-C environment was initialized for this ECB. All stack frames point to the next structure, which is the LWS.

```
*ISOC INITIAL STACK FRAME

01001964      00000000 00000000 BKP       00000000 80A17C34 R14       00A17D08 01001CB4           0071C930 01000264
   BCD                                                  @                  '                           I
01001984      01000250 0071C930           0700C198 01000200           00000000 00A17AB0           00300000 800F35CA
   BCD               &      I                  A
010019A4      01002FFF 00000000 R12       01001A2C 01001B3C NAB       00000000 00000000           00000000 00000000
010019C4      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
010019E4      00000000 0071C930           0071C496 00000038           00319E80 00319E80           00331560 003314B0
   BCD                      I                  D                                                         -
01001A04      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01001A24      00000000 00000000


*ISOC LWS

01001A2C      0A320070 58D09250           58E0D00C 982CD01C           0D1E0082 DF180000           00000000 00000000
01001A4C      00000000 00000000           00000000 00000000           01001B3C 00000000           00000000 00000000
01001A6C      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01001B2C      00000000 00000000           00000000 00000000
```

Figure 47. ISO-C Initial Stack Frame Dump

The remaining stack frames are formatted in reverse order. The first function called will be at the bottom and the last function called will be at the top. Each stack frame is labeled with the `#pragma map` name of the function.

Each stack frame contains a register save area. While reviewing the register contents, keep in mind that the compiler saves only those registers which are used by the function.

```
*ISOC STACK FRAME FUNCTION-called_function

01009F6C      10000000 01009EDC BKP       00000000 80A17E0A R14       00A17DB0 01009FFC           0071C930 01009F6C
   BCD                                                  =                  '                           I
01009F8C      80A17E04 00000000           00000000 00000000           00000000 00000000           00000000 00000000
   BCD             =
01009FAC      00000000 00000000 R12       01001A2C 01009FFC NAB       00000000 00000000           00000000 00000000
01009FCC      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01009FEC      00000000 00000000           00000000 00000000


   .
   .
   .

*ISOC STACK FRAME FUNCTION-called_function

01001BCC      10000A64 01001B3C BKP       10000000 80A17E0A R14       00A17DB0 01001C5C           0071C930 01001BCC
   BCD                                                  =                  '          *                I
01001BEC      80A17E04 000001F4           8087EA6A 00000070           00000000 00000000           01001CAC 00004650
   BCD             =          4                                                                                   &
01001C0C      000004B0 00000000 R12       01001A2C 01001C5C NAB       01001A2C 01001CB4           00000000 00000000
   BCD                                                    *
01001C2C      00000000 00000000           00000000 00000000           00000000 00000000           00000000 00000000
01001C4C      00000000 00000000           00000000 00000000


*ISOC STACK FRAME FUNCTION-qhph_first_function

01001B3C      10000000 01001964 BKP       00000000 80A17D70 R14       00A17DB0 01001BCC           0071C930 01001B3C
   BCD                                                  '                  '                           I
01001B5C      80A17D6A 0083E700           01000010 00851490           01000A64 01000A64           01000010 00000000
   BCD             '        X
01001B7C      01000010 00000000 R12       01001A2C 01001BCC NAB       00000000 00000000           00000000 00000000
01001B9C      00000000 00000000           00000000 00000000           00000000 00000000           00A17CBC 00000000
   BCD                                                                                                 @
01001BBC      00000170 00000000           00000000 00000000
```

Figure 48. ISO-C Stack Frame Function Dump

ISO-C static and heap areas follow the ISO-C stack frames and are identified by the DLM name.

```
*ISOC STATIC BLOCK
01802000    D8E9E9F4 FFFFFF00 0000007C 00000000 E3C8C9E2 40C9E240 E3C8C540 E2E3C1E3       QZZ4............THIS IS THE STAT
01802020    C9C340C4 C1E3C140 C2D3D6C3 D2404040 40404040 40404040 40404040 40404040       IC DATA BLOCK
01802040    40404040 40404040 4015F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0                .0000000000000000000000
01802060    F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0 F0F0F0F0                0000000000000000000000000000


*IN USE HEAP STORAGE
01801638    827F66C6 00000080 00000000 00000000 00000000 00000000 00000000 00000000       B..F............................
01801658    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000       ................................
018016B8    00000000 00000082                                                            .......B
```

Figure 49. ISO-C Static Block and Heap Storage Dump

If the error occurs in the ISO-C environment, registers 12 and 13 contain the following values:

R12 Address of ISO-C task communications area (TCA)

R13 Address of a stack frame.

Although the contents are not guaranteed, other ISO-C register conventions are:

R1 Parameter list pointer

R3 Function base address

R14 Return address

R15 Routine address or return value.

All other registers are in use by the compiler.

Brief Listing of Errors

The following list briefly describes an error condition and its probable cause.

Table 24. Brief Listing of Errors

Symptom	Probable Error
Unresolved Zxx VCONs during linkage editing	Including TARGET(TPF) segments in prelink or link-edit steps
Return code 4 from the prelinker	DLM call stubs not previously created
Unresolved VCONs to compiler components	Different levels of compiler and prelinker
System error CTL-3 - branch to location 0	DLM call stub either not created or not linked
System error CTL-3 - store into program	Program compiled with the NORENT parameter
Error during prelinking WARNING EDC4015: Unresolved references are detected: @@TRT Return code 4	The CTRT40 object module (OCO), found in the ACP.OBJ.RELvv PDS, must be copied to the ACP.CLIB.RELvv PDS and renamed to @@TRT.

Using C Function Trace

C function trace provides the ability to trace ISO-C programs. When an ISO-C program has been compiled with the TEST option of one of the C/370 family of compilers supported by the TPF system, C function trace provides the programmer with relevant information to expedite the analysis of C program problems. For information on how to use C function trace and sample trace output see *TPF Program Development Support Reference*.

Using Link Map Support for C Load Modules

The ZDMAP command allows you to display the link maps for C load modules to help you debug C load modules online. You must use both the offline C load module build tool (CBLD) and the offline loader (TPFLDR) if you want a C load module to have a link map. See "ISO-C Load Module Build Tool (CBLD)" on page 329 for more information about link map support in the C load module build tool.

The link map consists of a list of object files included in the C load module, a list of C function names in the object files, and the addresses of the object files and C functions.

Link map displays include both main storage addresses and the offsets of C functions into their respective object files. Any time an object file name or function name is displayed, the address of the object file or function is also displayed.

Parameters on the ZDMAP command allow you to request the following:

- A list of all object files in the C load module
- A list of all object files in the C load module and all functions contained in those object files
- A specific object file (or several object files whose names match input criteria)
- A specific object file (or several object files whose names match input criteria) and all functions contained in those object files
- A specific function (and the object file containing the function)
- The object file and function whose main storage addresses span a specified address.

See *TPF Operations* for more information about the ZDMAP command.

Customizing C/C++ Language Support

This chapter is a guide to customizing C language support on your TPF system. C language support is part of the base TPF system. See *TPF System Generation* for more information on how to build and customize a TPF system.

Required Customizations

You need to customize the TPF system to be able to take full advantage of C language support. This section includes a description of required and optional customizations.

TPF Globals

TPF globals are a unique phenomenon.

Most operating systems provide a means for application programs to access and modify variables in common storage. The structure of the TPF global areas, and the fact that they are not link-edited with the application load module, ensure that TPF application programs that manipulate the global areas will never be fully portable to other operating systems. However, there is a way for TPF application programs written in C to access the global areas.

This section describes the special concerns that arise when creating a C language interface to TPF globals, and the customization that you need to do before you can create application programs written in C that access TPF global fields and records. See the earlier section on globals for architecture and terminology.

- Assembler global tagnames are converted to C language globals by changing all @, $ or # characters to underscore (_). This is required because C restricts identifier names to alphanumerics and the _ character. This can introduce duplicate names. The user is responsible for resolving any name duplications.

  For example, consider assembler global tag @MAXCL, which defines the maximum CRT line count. This item has displacement into the global 1 area of X'44A', a length of 2 bytes, can be synchronized, and is a subsystem-unique global field. The following `#define` statement would appear in `c$globz.h` for this global:

  ```
  #define  _maxcl  0x044a0241
  ```

  The displacement is shown in bits 4–15 (X'44A'), the length in bits 16–23 (X'02'), attributes in bits 24–28 (B'01000'), and the global area where the item resides in bits 29–31 (B'001').
- To permit access to the IBM-supplied global fields, a version of `c$globz.h` is provided with C language support. However, because the nature and content of global areas and blocks varies from installation to installation, `c$globz.h` ***must be updated*** by the user to include installation-specific global tag names. When tags are added, deleted, or changed, `c$globz.h` must be modified and made available for use by application programs; the correct version of `c$globz.h` must be available at compile time. Similarly, C programs that refer to a given TPF global tag must be recompiled if there is a change made to any of the tag's attributes.

  TPF provides 2 sample utilities, GENTAG.C (for use with Assembler H) and GNTAGH.C (for use with HLASM), which demonstrate a means of automating the creation of `c$globz.h`. These programs can be run whenever changes have been made to the allocation of global fields or records. For more information about GENTAG and GNTAGH, see *TPF System Installation Support Reference*.

For a description of the TPF API functions that make use of the tag names to access or modify individual global fields or records, see the *TPF C/C++ Language Support User's Guide*.

The 6 ISO-C `glob` functions (`glob_keypoint`, `glob_modify`, `glob_lock`, `glob_update`, `glob_sync`, and `glob_unlock`) have several, similar requirements:

- They all must include. `tpfglbl.h` and `c$globz.h`, except for `glob_modify` which must include `tpfglobh` and `c$globz.h`.
- The argument coded as the globals or records must be defined in header file `c$globzh`. If they are not, results are unpredictable.
- These functions all have important considerations regarding giving up control because of locks or I/O considerations. Care must be taken when using them.

For details on these functions see their individual sections in the *TPF C/C++ Language Support User's Guide*.

Optional Customizations

This section includes a description of optional customizations.

Creating and Selecting Locales

This section describes the different types of locales.

What is a Locale?

A C *locale* consists of a set of constants that depend on geographic area. The constants affect the action of certain library functions (such as the string collation function `strcoll`) and control:

- Character editing and processing
- Monetary symbols and formatting
- Nonmonetary formatting
- Collating sequences
- Time zones.

Note: Time information is dependent on the system clock setting, which is assumed to be set to Greenwich Mean Time (GMT). If you set the system clock to something other than GMT, see *TPF System Generation* for details on how to customize the clock value.

The `setlocale` function selects the appropriate portion of the program's locale. It can be used to change or query the program's entire current locale, or portions of it.

The basic format of the function is:

```
setlocale(category, locale)
```

where **category** specifies which attributes of the locale you want to change or query, and **locale** identifies the locale itself (as in LC_C_GERMANY, LC_C_SPAIN, and so on). If you specify ″″ for **locale**, the `setlocale` function will default the locale to `LC_DEFAULT_LOCALE`.

When the first C program for a given ECB is activated, the equivalent of

```
setlocale(LC_ALL, "C");
```

has been performed. A value of C for the locale specifies the minimal environment for C translation. The program can then call the `setlocale` function if a different locale is desired.

The `setlocale` function is described in more detail in the library reference for the IBM C or C++ compiler on the System/390 platform used by your installation.

In ISO-C the C locale supplied by IBM is in C$S370 which is linked in library CISO. All the other shipped locales are in DLMs:

Locale	Source	DLM
LC_C_GERMANY	CLLGER	CLLG
LC_C_FRANCE	CLLFRN	CLLF
LC_C_UK	CLLENG	CLLE
LC_C_ITALY	CLLITL	CLLI
LC_C_SPAIN	CLLSPA	CLLS
LC_C_USA	CLLUSA	CLLU
LC_C_TPF	CLLTPF	CLLT

These locales are defined by calling the EDCLOC macro. All of the shipped locales specify a TZDIFF value of 1500 (that is, use the system time zone difference). The TPF system time zone difference is defined in keypoint record A at tag CK1LGD. In order for the standard C time formatting functions to use this value you must define the appropriate time zone name (TNAME). To handle daylight savings time you must also specify the following EDCLOC parameters: DSTSTM, DSTSTW, DSTSTD, STARTTM, SHIFT, DSTENM, DSTENW, DSTEND, ENDTM, and DSTNAME.

For TARGET(TPF), all of the locales are defined in CP segment CL04 (CCLANG). For TARGET(TPF) the absolute value of TZDIFF must not exceed 1440 (minutes, that is 24 hours). System time zone differences are not supported for TARGET(TPF). All of the TARGET(TPF) locales specify a TDZIFF value of -300 for North American eastern standard time. You should modify these locales as appropriate for your installation.

Creating a New ISO-C Locale

You can create a new locale, or modify an existing one, by specifying a new set of parameters to the assembler macro EDCLOC. Edit the locale source file (xxx) and change the EDCLOC parameters to make small changes in a default locale. New locales are created by:

- Coding an ISO-C compatible assembler program with a call to the EDCLOC macro, because each locale must be in its own dynamic load module (DLM). The first 4 characters of the name of the DLM being created, the name of the locale, and the label on the EDCLOC macro must all be the same.
- Creating a build script for the new DLM containing a DLM record and a record indicating the name of the locale definition file.
- Following the usual procedure for assembling, building, allocating, and loading a DLM.

After the new DLM is loaded online, the new values in the new locale are available to the functions `setlocale` and `localeconv`.

For example, suppose we wanted to set up a new locale for the North Pole. The locale definition could be as in Figure 50 on page 304. At the North Pole we might find that every day is Saturday and that every month is December.

```
************************************************************************
* MODULE NAME: NPOLLC                                                  *
* DESCRIPTION: DEFINITION OF C LOCALE "NPOL" FOR THE NORTH POLE.       *
*                                                                      *
************************************************************************
NPOL   EDCLOC  CHARTYP=1,CTYPE=,CTYPE1=,UPPER=,LOWER=,COLLTAB=,
               COLLSTR=,DEC='.',SEP=,GROUP=(0,0),ICURR=,CURR=,
               MDEC=,MSEP=,MPLUS=,MMINUS=,
               MIFDIGITS=CHAR_MAX,
               MFDIGITS=CHAR_MAX,MGROUP=(0,0),MPCSP=CHAR_MAX,
               MPSBYS=CHAR_MAX,MNCSP=CHAR_MAX,MNSBS=CHAR_MAX,
               MPLUSPOS=CHAR_MAX,MMINUSPOS=CHAR_MAX,
               SDAYS=(SAT,SAT,SAT,SAT,SAT,SAT,SAT),
               LDAYS=(SATURDAY,SATURDAY,SATURDAY,SATURDAY,SATURDAY,
               SATURDAY,SATURDAY),
               SMONS=(DEC,DEC,DEC,DEC,DEC,DEC,DEC,DEC,DEC,DEC,DEC,
               DEC),
               LMONS=(DECEMBER,DECEMBER,DECEMBER,DECEMBER,DECEMBER,
               DECEMBER,DECEMBER,DECEMBER,DECEMBER,DECEMBER,
               DECEMBER,DECEMBER),
               DATFMT='%m/%d/%y',TIMFMT='%H.%M.%S',AM='AM',PM='PM',
               DATTIM='%m/%d/%y %X',
               TZDIFF=0,TNAME=NPST,
               DSTSTM=0,DSTSTW=0,DSTSTD=0,STARTTM=0,SHIFT=0,
               DSTENM=0,DSTENW=0,DSTEND=0,ENDTM=0,DSTNAME=,
               VERSION=1

     END  ,
```

Figure 50. Assembler Program for a New Locale

This assembler program only contains a call to the EDCLOC macro (and END). No calls to the BEGIN or TMSPC macro are required; EDCLOC generates all of the required code.

A build script for the North Pole locale could be

```
########################################################################
# LOAD MODULE NAME: NPOL                                               #
# DESCRIPTION: North Pole locale "NPOL" definition.                    #
########################################################################
DLM NPOL41
NPOLLC41
```

Figure 51. Build Script for a New Locale

The DLM locale is assembled, built, allocated, and loaded. An application can call

```
      setlocale(LC_ALL, "NPOL") ;
```

and all standard library functions that take locale values into account would operate in a manner appropriate to the North Pole.

The parameters for defining a locale are fully described in the user's guide for the IBM C or C++ compiler on the System/390 platform used by your installation. Keep the following in mind when creating a locale for the TARGET(TPF) environment:

- The TPF version of the locale generator macro will not allow you to specify CHARTYP greater than 1; that is, multiple-byte character sets are not supported.
- It is not possible to code TZDIFF with a value greater than 1440. On other systems this will call the *system* time zone difference; this is not supported for TPF systems.

See the IBM C or C++ user’s guide and programming guide for the IBM System/390 platform used by your installation for a description of the macro parameters used to define a locale.

Creating a New localedef Utility-Based Locale

localedef utility-based locales must co-exist with the existing EDCLOC macro-based locales. The existing EDCLOC locales are coded in the application by their load module name. localedef utility-based locales are coded by their 4-character internal name or the long external name. On the `setlocale`() function, the locale name is first assumed to be a localedef utility-based locale. If a localedef utility-based locale module is not found mapping to the locale name coded, an attempt will then be made to load it as an existing EDCLOC locale.

Preparing the EDCLDEF JCL Procedure for the TPF System: The EDCLDEF job control language (JCL) procedure provided with the compiler takes a locale definition as input and displays the locale module. For the TPF system, the module needs TPF startup code. You must modify the EDCLDEF JCL procedure to create object code for the locale, and the object code can then be link-edited as a new locale module. Figure 52 on page 306 shows the EDCLDEF JCL procedure as it is shipped. Figure 53 on page 308 shows you a sample of the EDCLDEF JCL procedure after it has been modified. In these figures, changes that you must make to the EDCLDEF JCL procedure are shown in a **`bold example`** font; for example **`SYSLBLK='3200`**’.

You must make the following changes to the EDCLDEF JCL procedure:

- Remove the SYSLBLK= line because it is no longer needed.
- Replace the OUTFILE definition with OUTFILE=. OUTFILE will be used for placing object code generated from the compile step. The data set will be a partitioned data set (PDS) provided in the JCL using the procedure with DISP=SHR.
- Add CREATE and MERGE steps after the LOCALDEF step to modify the locale C source.

  Note: The CREATE step requires the PRAGMA segment, which you must create and place in your PDS. For example, the EDCLDEF JCL procedure found in Figure 52 on page 306 has a PDS named ACP.CHDR.SARAT(PRAGMA). The PRAGMA will contain one line:

  ```
  #pragma nomargins nosequence
  ```

  In the JCL that uses this procedure, code MARGINS(1,72) on the CPARM parameter. This, with the modification to locale C source, will allow you to compile the >80-record-size locale source with the TPF fixed 80-record-size header files.
- Remove the last two steps, INCLUDE and LKED, and change the data set for output of the COMPILE step (SYSLIN) to OUTFILE. See Figure 52 on page 306 for an original EDCLDEF procedure as shipped with the compiler, and Figure 53 on page 308 for a modified EDCLDEF procedure for the TPF system.

```
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5647-A01                                                           *
//* (C) COPYRIGHT IBM CORP. 1988, 1997 ALL RIGHTS RESERVED             *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA ADP                    *
//* SCHEDULE CONTRACT WITH IBM CORP                                    *
//*                                                                    *
//**********************************************************************
//***********************************************************/
//*    MVS PROGRAM PRODUCTS TEAM                            */
//*                                                         */
//* THE FOLLOWING CHANGES MADE TO CORRECT PROCEDURE         */
//* 1. VIO CHANGED TO SYSDA                                */
//* 2. LIBPRFX='CEE' TO LIBPREX='SYS1.CEE.V2R4M0'          */
//* 3. LNGPRFX='CBC' TO LNGPRFX='SYS1.CBC.V2R4MO'          */
//*                                                        */
//**********************************************************/
//**********************************************************************
//*                                                                    *
//* INVOKE THE LOCALEDEF UTILITY OT CREATE C SOURCE CODE               *
//* THEN COMPILE AND LIKN EDIT THE PROGRAM                             *
//*                                                                    *
//* OS/390 C/C++                                                       *
//*                                                                    *
//* RELEASE LEVEL:  02.04.00  (VERSION.RELEASE.MODIFICATION LEVEL)     *
//*                                                                    *
//**********************************************************************
//*
//EDCLDEF PROC  INFILE=,                  < INPUT ... REQUIRED
//  CREGSIZ='4M',                         < COMPILER REGION SIZE
//  CPARM=,                               < COMPILER OPTIONS
//  SYSLBLK='3200',                      < BLOCKSIZE FOR &&LOADSET
//* LIBPRFX='CEE';                        < PREFIX FOR LIBRARY DSN
//  LIBPRFX='SYS1.CEE.V2R4M0',            < PREFIX FOR LIBRARY DSN
//* LNGPRFX='CBC',                        < PREFIX FOR LANGUAGE DSN
//  LNGPRFX='SYS1.CBC.V2R4M0',            < PREFIX FOR LANGUAGE DSN
//  CLANG='EDCMSGE', < NOT USED IN THIS RELEASE. KEPT FOR COMPATIBILITY
//  LOPT=,                                < LOCALDEF OPTIONS
//  DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200',      <DCB FOR LRECL 80
//  DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800(', <DCB FOR LRECL 3200b
// OUTFILE='&&GSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,SPACE=(TRK,(7,7,1))',
//  TUNIT='SYSDA'                           < UNIT FOR TEMPORARY FILES
//*
//*-------------------------------------------------------------
//* LOCALDEF STEP:
//* INVOKE CBC3LDEF MODULE TO READ LOCALE DEFINITION FILE AND
//* GENERATE C CODE.
//*-------------------------------------------------------------
//          EXEC PGM=CBC3LDEF,REGION=6144K,
//    PARM=(&LOPT)
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//         DD  DSNAME=&LNGPRFX..SCBCCMP,DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DSNAME=&INFILE,DISP=SHR
//SYSPUNCH DD  DSNAME=&&SYSCIN,DISP=(NEW,PASS),
//         DCB=(RECFM=FB,LRECL=120,BLKSIZE=4800),
//         SPACE=(4800,(100,100)),UNIT=&TUNIT.
//EDCCMAP  DD  DSNAME=&LIBPRFX..SCEEMAP,DISP=SHR
//EDCLOCL  DD  DSNAME=&LIBPRFX..SCEELOCL,DISP=SHR
```

Figure 52. An EDCLDEF JCL Procedure as Shipped with the Compiler (Part 1 of 2)

```
//*-------------------------------------------------------------
//*  COMPILE STEP:
//*-------------------------------------------------------------
//COMPILE EXEC PGM=CBCDRVR,REGION=&CREGSIZ,
//   COND=(4,LT,LOCALDEF),
//   PARM=('NOSEQ,NOMARGINS',
//   '&CPARM')
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//         DD  DSNAME=&LNGPRFX..SCBCCMP,DISP=SHR
//SYSMSGS  DD  DUMMY,DSN=&LNGPRFX..SCBC3MSG(&CLANG),DISP=SHR
//SYSLIN   DD  DSNAME=&&SYSCIN,DISP=(OLD,DELETE)
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEEH.H,DISP=SHR
//         DD  DSNAME=&&LOADSET,UNIT=&TUNIT.,
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSUT1   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT4   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
```

Figure 52. An EDCLDEF JCL Procedure as Shipped with the Compiler (Part 2 of 2)

```
//*********************************************************************
//*                                                                   *
//* LICENSED MATERIALS - PROPERTY OF IBM                              *
//*                                                                   *
//* 5647-A01                                                          *
//* (C) COPYRIGHT IBM CORP. 1988, 1997 ALL RIGHTS RESERVED            *
//*                                                                   *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                      *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA ADP                   *
//* SCHEDULE CONTRACT WITH IBM CORP                                   *
//*                                                                   *
//*********************************************************************
//**********************************************************/
//*    MVS PROGRAM PRODUCTS TEAM                             */
//*                                                        */
//* THE FOLLOWING CHANGES MADE TO CORRECT PROCEDURE         */
//* 1. VIO CHANGED TO SYSDA                                */
//* 2. LIBPRFX='CEE' TO LIBPREX='SYS1.CEE.V2R4M0'          */
//* 3. LNGPRFX='CBC' TO LNGPRFX='SYS1.CBC.V2R4MO'          */
//*                                                        */
//**********************************************************/
//*********************************************************************
//*                                                                   *
//* INVOKE THE LOCALEDEF UTILITY OT CREATE C SOURCE CODE              *
//* THEN COMPILE AND LIKN EDIT THE PROGRAM                            *
//*                                                                   *
//* OS/390 C/C++                                                      *
//*                                                                   *
//* RELEASE LEVEL:  02.04.00  (VERSION.RELEASE.MODIFICATION LEVEL)    *
//*                                                                   *
//*********************************************************************
//*
//EDCLDEF PROC  INFILE=,                  < INPUT ... REQUIRED
//  CREGSIZ='4M',                        < COMPILER REGION SIZE
//  CPARM=,                              < COMPILER OPTIONS
//* LIBPRFX='CEE',                       < PREFIX FOR LIBRARY DSN
//  LIBPRFX='SYS1.CEE',                  < PREFIX FOR LIBRARY DSN
//* LNGPRFX='CBC',                       < PREFIX FOR LANGUAGE DSN
//  LNGPRFX='SYS1.CBC',                  < PREFIX FOR LANGUAGE DSN
//  CLANG='EDCMSGE', < NOT USED IN THIS RELEASE. KEPT FOR COMPATIBILITY
//  LOPT=,                               < LOCALDEF OPTIONS
//  DCB80='(RECFM=FB,LRECL=80,BLKSIZE=3200)',      <DCB FOR LRECL 80
//  DCB3200='(RECFM=FB,LRECL=3200,BLKSIZE=12800)', <DCB FOR LRECL 3200
//  OUTFILE=,
//  TUNIT='SYSDA'                        < UNIT FOR TEMPORARY FILES
//*
//*--------------------------------------------------------------
//* LOCALDEF STEP:
//* INVOKE CBC3LDEF MODULE TO READ LOCALE DEFINITION FILE AND
//* GENERATE C CODE.
//*--------------------------------------------------------------
//LOCALDEF  EXEC PGM=CB3LDEF,REGION=6144K,
//    PARM=(&LOPT)
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//         DD  DSNAME=&LNGPRFX..SCBCCMP,DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DSNAME=&INFILE,DISP=SHR
//SYSPUNCH DD  DSNAME=&&SYSCIN,DISP=(NEW,PASS),
//         DCB=(RECFM=FB,LRECL=120,BLKSIZE=4800),
//         SPACE=(9800,(500,200),UNIT=&TUNIT.
//EDCCMAP  DD  DSNAME=&LIBPRFX..SCEECMAP,DISP=SHR
//*EDCLOCL  DD  DSNAME=&LIBPRFX..SCEELOCX,DISP=SHR
//EDCLOCL  DD  DSNAME=&LIBPRFX..SCEELOCL,DISP=SHR
//*
```

Figure 53. A Sample Modified EDCLDEF JCL Procedure (Part 1 of 2)

```
//*-------------------------------------------------------------
//*  CREATE SYSCIN1
//*-------------------------------------------------------------
//CREATE EXEC PGM=ICEGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1   DDDSN=ACP.CHDR.SARAT(PRAGMA),DISP=SHR
//SYSUT2   DD  DSNAME=&&SYSCIN1,DISP=(NEW,PASS),
//         DCB=(RECFM=FB,LRECL=120,BLKSIZE=4800),
//         SPACE=(10,(10,10)),UNIT=SYSDA
//SYSIN    DD DUMMY
//*
//*----------------------------------------------
//*  MERGE SYSHD AND SYSCIN PRODUCING SYSCIN2
//*----------------------------------------------
//MERGE EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=&&SYSCIN1,DIS=(OLD,DELETE)
//         DD DSN=&&SYSCIN,DIS=(OLD,DELETE)
//SYSUT2   DD  DSNAME=&&SYSCIN2,DISP=(NEW,PASS),
//         DCB=(RECFM=FB,LRECL=120,BLKSIZE=4800),
//         SPACE=(4000,(100,100)),UNIT=SYSDA
//SYSIN    DD DUMMY
//*-------------------------------------------------------------
//*  COMPILE STEP:
//*-------------------------------------------------------------
//COMPILE EXEC PGM=CBCDRVR,REGION=&CREGSIZ,
//   COND=(4,LT,LOCALDEF),
//   PARM=('NOSEQ,NOMARGINS',
//   '&CPARM)
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,DISP=SHR
//         DD  DSNAME=&LNGPRFX..SCBCCMP,DISP=SHR
//SYSMSGS  DD  DUMMY,DSN=&LNGPRFX..SCBC3MSG(&CLANG),DISP=SHR
//SYSIN    DD  DSNAME=&&SYSCIN2,DISP=(OLD,DELETE)
//SYSLIB   DD
//         DD  DSNAME=&LIBPRFX..SCEEH.H,DISP=SHR
//         DD  DSNAME=&LIBPRFX..SCEEH.SYS.H,DISP=SHR
//SYSLIN   DD  DSNAME=&OUTFILE,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSUT1   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT4   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB80
//SYSUT5   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT6   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT7   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT8   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),DCB=&DCB3200
//SYSUT9   DD  UNIT=&TUNIT.,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
```

Figure 53. A Sample Modified EDCLDEF JCL Procedure (Part 2 of 2)

Place the modified EDCLDEF JCL procedure in the proclib data set. See Figure 54 on page 310 for a sample JCL to create object code for a locale using the modified EDCLDEF JCL procedure. In this figure, changes that you must make to the EDCLDEF JCL procedure shipped are shown in a **`bold example`** font; for example **`(EDC$1TEY)`**.

```
//LCDEFC   JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A
//* NOTE: EDCLDEFT isthe modified EDCLDEF procedure.
//LDEFCOMP EXEC EDCLDEFT,
//* provide PDS member where the locale definition source
//* is placed for INFILE parameter.
//        INFILE='ACP.LOCALE.SOURCE(EDC$1TEY)',
//* NOTE: ProvidePDS member where the object code for the locale
//* must be placed for OUTFILE parameter
//        OUTFILE='ACP.REL40.OB(CL1TEYVV)',
//* NOTE: In addition to any other compile options always provide
//* MARGINS(1,72) on CPARM parameter.
//        CPARM='MARGINS(1,72)'
//COMPILE.SYSLIB DD DSNAME=ACP.CHDR.REL40,DISP=SHR
/*
//
```

Figure 54. Sample JCL Using the Modified EDCLDEF JCL Procedure

Preparing localedef Utility-Based Locale Load Modules: You can use one of the locale definitions provided with the compiler, modify one of them, or create a new one using the format documented in the IBM C/C++ programming guide on the IBM System/390 platform used by your installation.

To Prepare the localedef Utility-Based Locales Load Module::

1. Prepare the locale definition. See the IBM C/C++ programmer's guide on the IBM System/390 platform used by your installation.
2. Run the modified EDCLDEF JCL procedure to create the object file (TEXT) for the locale.
3. Create the build script for the locale load module. These locales must be link-edited with the additional CENTPT40 object file. See Figure 55 on page 311 for an example of a build script for the 1TEY locale compiled in JCL. In this figure, changes that you must make to the EDCLDEF JCL procedure shipped by IBM are shown in a **`bold example`** font; for example **`PDS member CL1TEY`**

```
####################################################################
# This build script is to build CNLT locale module.                #
# The object code for this locale is in PDS member CL1TEY.#
# The locales external long name is MYLOC.IBM-1047, the internal    #
# short name is 1TEY where 1T maps to MYLOC from locale name table  #
# and EY maps to IBM-1047 from code set name table.  The locale     #
# module will reside in CNLT and the locales internal name          #
# 1TEY should be mapped to CNLT in locale module name mapping table #
# in CLM0DN.                                                        #
####################################################################
#                                                                  #
# SCRIPT NAME:   CNLTBS                                            #
#                                                                  #
# DESCRIPTION:  New extended locale called MYLOC that uses         #
#               code set name IBM-1047 or <CC> = EY.               #
#               Locale code or <LT> code = 1T.                     #
#               Remember <LT> for user defined locales need to     #
#               have a numeric.                                    #
####################################################################
DLM  CNLT40

#Object File  Function                                Source Language
#-----------  --------                                ---------------

CENTPT40      # remember this is needed for           Assembler
              # locales
CL1TEYVV      # TEXT created by the modified          Locale definition
           # EDCLDEF PROC
```

Figure 55. Example of a Build Script for the 1TEY Locale

4. Using the C load module build tool (CBLD), create the locale load module.

Other Changes Needed to Use localedef Utility-Based Locales: Add an entry for the locale in the locale name table by adding an EDCLOCNM macro call as documented in the IBM C/C++ programming guide for the compiler on the IBM System/390 platform used by your installation. The locale name table is in the CLNM assembler (BAL) source file. As shipped, this would have the EDCLOCNM call entries for the localedef utility-based locales shipped with the TPF system. Locale name table segment CLNM is part of DLM CLNM. The following is an entry that would be added for the sample locale:

```
EDCLOCNM TYPE=ENTRY,LOCALE='MYLOC',CODESET='IBM-1047',CODE='1T'
```

If a new code set is used that is not in the code set name table, you must add an entry for it. Code set name table is in the CSNM assembler (BAL) source file. Code set name table CSNM is part of DLM CSNM. See “Character Sets” on page 314 for more information about the CSNM code set name table.

The CMLM source file contains a table mapping the localedef utility-based locale names to the TPF locale load module names. This source file is part of DLM CMLM. Modify the CMLM source file to add an entry mapping the new locale name to the load module name. The locale name has two parts:

- The first part is the 2-character Language Territory code, <LT>. If the locale definition is not an IBM-shipped definition, the first character must be numeric.
- The second part is the 2-character code, <CC>, which identifies the CodeSetRegistry-CodeSetEncoding. Use the <CC> code from the code set names and the CC code table in the IBM C/C++ programming guide for the compiler on the IBM System/390 platform used by your installation. For the sample locale, the entry coded would be:

```
["1TEY","CNLT"],
```

Using localedef Utility-Based Locales: You can code the new localedef utility-based locale name two different ways in the `setlocale` function call:

- locale descriptive name

  The descriptive name is the concatenation of the LOCALE and CODESET parameter values separated by a period (.) on the EDCLOCNM macro call entry for the locale in the locale name table:

  ```
  /* Using the descriptive name */
  setlocale(LC_ALL, "MYLOC.IBM-1047");
  ```
- short internal name

  The internal name is the concatenation of the <LT> and <CC> codes for the locale:

  ```
  /* Using the short internal name */
  setlocale(LC_ALL, "1TEY");
  ```

Before activating any application that uses the locale, do the following:

1. Modify locale name table CLNM.
2. Modify code set name table CSNM.
3. Modify locale module name mapping table CMLM as in Other Changes Needed to Use localedef Utility-Based Locales on page 311.
4. Build and load the CLNM, CSNM, and CMLM DLMs.
5. Build the locale load module.

See the IBM C/C++ programming guide and user’s guide for the compiler on the IBM System/390 platform that is used by your installation for more information about the localedef utility-based locale definitions and how to use, customize, and modify them.

For **setlocale(LC_ALL,″″)**, the C locale is set. You can change this definition by setting the locale-related environment variable. For example, if **setenv(″LC_ALL″,″MYLOC.IBM-1047″,1)** is issued to initialize the LC_ALL environment variable, **setlocale(LC_ALL,″″)** will then set and return the ″MYLOC.IBM-147″ locale.

Note: This indicates that you have changed the default NULL string locale set by the TPF system.

Creating a New TARGET(TPF) Locale

For TPF systems, a set of C locales is provided in copy member CL04 of the CCLANG CSECT. A locale is generated using a special assembler macro to which you pass parameters. The locale in effect when the first C program is activated for a given ECB is known as EDC$S370 in CL04; this is the *system name* for the locale shown in the previous example. For other locales the system name appearing in CL04 has EDC$ as the first part of the name followed by *xxxx* (EDC$*xxxx*), where *xxxx* is the first 4 characters of the name of the country. For example, the system name for the locale for Germany is EDC$GERM.

Note: You can find that the EDC$S370 locale is not an acceptable locale for any of the applications in your environment. If this is true you can simply modify the macro parameters for this locale in CL04, and reassemble CCLANG. This frees application programs from always having to call the `setlocale` function before initiating their normal processing.

You can create a new locale, or modify an existing one, by specifying a new set of parameters to the assembler macros in CL04. To do this, modify CL04 as follows:

1. Copy 1 of the existing locales that describes a locale most similar to the 1 that you are creating. Choose a system name for the new locale with EDC$ as the first 4 characters followed by *xxxx* (EDC$*xxxx*) where *xxxx* represents 1–4 characters.
2. Change the invocation parameters to define the categories of the new locale.
3. Add an entry to the LOCALTAB table in CL04 for your new locale. (Follow the instructions in the listing commentary.)
4. Reassemble CCLANG and link-edit the control program again. Load the new control program to your system. Your new locale is now available for use by applications. It can be called (via `setlocale`) using the *xxxx* portion of the locale name defined in step 1.

C Language Support User Exits

The C language user exits are dynamic user exits that can be activated the first time a C program is called for a given ECB. When the exits are given control, the first C language stack block has been allocated, and the first stack frame, containing the library function work area, has been initialized. Information concerning the C user exit interface can be found in the *TPF System Installation Support Reference*.

User exits are provided, as follows:

1. The first user exit (CSK, CENV) provides the system programmer the capability to make changes to the environment.
2. Other user exits support test tools to allow trap points before and after a library function is called.
3. Two user exits provide access to entry and exit through stack processing.

Extending the Library Function Work Area

The purpose of the library function work area is to provide an area of storage for data that must be available to various library functions throughout the life of an ECB. User-written library functions can also require storage of this type in order to perform their tasks.

The C user exit can be used to allocate and initialize additional storage in the library function work area for use by library functions. The storage is allocated in the first stack frame, beginning at the end of the IBM library function work area. This region is known as the user expansion area. The pointer to the user expansion area should be loaded by the user library function whenever access to the storage is desired. See Coding Your Own Library Functions, for an example.

Collection of Data

User exits (EFCE and RTNE) can be activated every time a C program is called for a given ECB, so the number of times the user exit is called in a given time interval can be accumulated. This can be used to calculate the percentage of ECBs active in the system, over a period of time, that called C programs.

Customizing User Data Area in ISO-C Modules

The TPF system provides user data areas that can be initialized with user-defined data at system startup. You can access these user data areas from applications written in C/C++ by using the data object name for the user data area in the desired module type. Following is the TPF code segment in which the user data area resides and the user data area data object name for each module type:

Module Type	Startup Code Segment	Data Object Name
DLM	CSTRTD	@@USRDMD
LLM	CSTRTL	@@USRLBD
DLL	CSTDLL	@@USRDLD

Object code only (OCO) dummy stub data objects are provided to set up and initialize the user data areas. These dummy stub data objects consist of the following code, where @@USRxxx is the data object name:

```
@@USRxxx CSECT
@@USRxxx AMODE ANY
@@USRxxx RMODE ANY
         DS    0D
         DC    4F'0'
         END
```

Note: These stub data objects are autocalled into the module.

You supply your own user data area by replacing the particular existing OCO dummy stub data object in data set ACP.STUB.REL40 with a user data object by using the following steps:

1. Create a source segment containing the code for the stub data object you wish to customize.
2. Substitute your customized data in place of the current data.
3. Assemble the segment with data set ACP.STUB.REL40 as the target for the object module.
4. Link-edit all modules requiring access to the user data area.

Note: To avoid loading incorrect user data, there should be a single copy of that particular stub data object and it should be in data set ACP.STUB.REL40. Delete any copies of the stub data object that are in data set ACP.CLIB.REL40.

Character Sets

You must complete a number of offline tasks to create support for a new character set on the online system.

Choosing a New Character Set

Character sets are chosen on the basis of the kinds of letters and symbols required. Once these requirements are understood, you can choose the appropriate character sets. To learn more about character sets, see *Character Data Representation Architecture Reference and Registry*.

Translating Character Sets

A character set is referred to by a number called a *coded character set identifier* (CCSID). The TPF system contains characters in one or more CCSIDs (or TPFCCSIDs).

When the CCSIDs of the TPF system matches another system, no translation is necessary. If the character sets are different, a translation mechanism must exist to transform each character from the remote CCSID to a corresponding character of the TPF system CCSID.

Suppose the TPF system uses CCSID 500 and the RS/6000 uses CCSID 819. To communicate with single-byte TPF system using 500, there must be translations between 819 and 500 available: 819 to 500 for the TPF system and 500 to 819 for the remote server.

CCSIDs that IBM specifies are in the CSNM table. Code set name table CSNM is part of the DLM CSNM. This table joins aspects of CCSIDs through calls to the CSNMC macro. The code set name for a CCSID is the name for the CCSID in the *Character Data Representation Architecture Reference and Registry*, the CDRA code set registry. The CCSID itself is often the numerical part of the registry name. Each CCSID has a two-letter code associated with it. This code is used with other two-letter codes to specify a translation. The STYLE parameter of the CSNMC macro indicates whether the CCSID is single-byte (S), double-byte (D), or mixed-byte (M). The SINGLE and DOUBLE parameters are required for mixed-byte CCSIDs. These parameters indicate the CCSIDs for the single-byte and double-byte components of a mixed-byte CCSID.

```
CSNMC CODESET='IBM-037',CODE='EA',CCSID=037,STYLE=S
CSNMC CODESET='IBM-284',CODE='EJ',CCSID=284,STYLE=S
CSNMC CODESET='IBM-500',CODE='EO',CCSID=500,STYLE=S
CSNMC CODESET='ISO8859-1',CODE='I1',CCSID=819,STYLE=S
CSNMC CODESET='IBM-850',CODE='AA',CCSID=850,STYLE=S
CSNMC CODESET='IBM-1027',CODE='EX',CCSID=1027,STYLE=S
CSNMC CODESET='IBM-1047',CODE='EY',CCSID=1047,STYLE=S

CSNMC CODESET='IBM-300',CODE='EN',CCSID=300,STYLE=D
CSNMC CODESET='IBM-300',CODE='EN',CCSID=4396,STYLE=D

CSNMC CODESET='IBM-930',CODE='EU',CCSID=930,STYLE=M,
      SINGLE=290,DOUBLE=300
CSNMC CODESET='IBM-932',CODE='AB',CCSID=932,STYLE=M,
      SINGLE=897,DOUBLE=301
CSNMC CODESET='IBM-939',CODE='EV',CCSID=5035,STYLE=M,
      SINGLE=1027,DOUBLE=300
```

Figure 56. An Example of the CSNM Table of CCSIDs

Picking CCSID 500 for the TPF system CCSID and 819 for the RS/6000 system, for example, requires a translation table called EOI1 (500 to 819). The translation is done on inbound data and no translation is done on outbound data. Therefore, the TPF system needs the I1EO table.

Changes to the CSNM table are not needed unless you are adding a CCSID for a code page that is entirely new.

Defining a Translation

Translations are defined using the CSNM table by specifying the code set registry name, the two-letter code, the CCSID, and the STYLE parameter. A translation table is required that corresponds to this CCSID specification.

The tables for many CCSIDs are shipped as a part of the IBM C language support product. The name of the code page for the character set consists of the product identifier, EDCU, followed by the two-letter code of the character set. Two code page tables are joined to create a translation table. The names of the translation tables provided by IBM C language are of the form EDCU*xxyy*, where *xx* is the two-letter code for the TPF system table and *yy* is the two-letter code for the remote table. These tables are used as input to the GENXLT program.

Translating on a TPF System

On a TPF system `iconv` runs online. This requires the translation table it uses to be constructed in the form of a DLM and that its name conform to TPF naming standards.

The EDCGNXLT process uses TPF code sets and the remote code sets to create translation tables. Figure 57 shows a part of the EDCUEOI1 GENXLT translation table for IBM-500 to ISO8859-1 CCSID 819.

```
0x00    0x00   <NUL>
0x01    0x01   <SOH>
0x02    0x02   <STX>
0x03    0x03   <ETX>
0x04    0x9c   <SEL>
0x05    0x09   <tab>
0x06    0x86   <RNL>
0x07    0x7f   <DEL>
0x08    0x97   <GE>
0x09    0x8d   <SPS>
0x0a    0x8e   <RPT>
0x0b    0x0b   <vertical-tab>
0x0c    0x0c   <form-feed>
0x0d    0x0d   <carriage-return>

⋮

0x7f    0x22   <quotation-mark>
0x80    0xd8   <O-slash>
0x81    0x61   <a>
0x82    0x62   <b>
0x83    0x63   <c>
0x84    0x64   <d>
0x85    0x65   <e>
0x86    0x66   <f>
0x87    0x67   <g>
0x88    0x68   <h>
0x89    0x69   <i>
0x8a    0xab   <left-angle-quotes>
0x8b    0xbb   <right-angle-quotes>
0x8c    0xf0   <eth>
0x8d    0xfd   <y-acute>
0x8e    0xfe   <thorn>
0x8f    0xb1   <plus-minus>
0x90    0xb0   <degree>
0x91    0x6a   <j>
0x92    0x6b   <k>
0x93    0x6c   <l>

⋮
```

Figure 57. An Example of a GENXLT Translation Table

The EDUGNXLT program generates a translation table which must be built into a DLM for online use. The IBM-supplied translation DLMs are called CPG*x* (where *x* is 0, 1, ..., 9). (User-supplied translation DLMs can be used in the JCL that runs the ECUGNXLT job as well.) The name of the CPG*x* (or user-supplied) DLM is paired with the original set of two-letter codes and placed in a table in a segment called CPGS. For example, in the IBM-supplied table, I1EO is paired with CPG0, AAEY with CPG1, and ABEU with CPG9. These form an *xxyy* pair that is used at run time to find the CPG*x* name of the DLM that is required to perform the translation. The pairings for new translation tables are added to this table; CPGS is recompiled, linked, and loaded, along with the new DLM that contains the translation tables.

Keeping CCSIDs Compatible

When deciding on mixed-byte character sets, it is critical to use CCSIDs that are compatible. Just because a given CCSID refers to a single-byte character set, it does not mean that the CCSID can be used wherever a single-byte character set is needed. For example, consider Table 25, which displays compatible Japanese to ASCII or EBCDIC CCSIDs.

Table 25. Compatible Single-, Double-, and Mixed-Byte CCSIDs

Character Set	Single CCSID	Double CCSID	Mixed CCSID
Japanese to ASCII	897	301	932
Japanese to EBCDIC	290	300	930

The 897 CCSID cannot be used in place of 290 because 897 is not compatible with the 930 CCSID. The 930 CCSID is constructed so that it is compatible with 290, not with 897. If you use incompatible CCSIDs, connections from the TPF system to remote systems may be refused. The purpose of the SINGLE and DOUBLE parameters is to help keep the component CCSIDs consistent. It is the user's responsibility to define mixed-byte CCSIDs with the CSNMC macro using single-byte and double-byte CCSIDs associated with the mixed-byte CCSIDs. For more information see *Character Data Representation Architecture Reference and Registry*.

Creating a Translation Table

Some sites may require code sets in addition to those provided with the TPF system product. Information and procedures for additional code sets are obtained from the MVS Language Environment (LE) product.

The first step is to identify the additional code sets that you need. To do this, review the following publications for the characteristics of the various code sets that are available and select compatible code sets.

- *Character Data Representation Architecture Reference and Registry* (called the CDRA Registry)
- *Language Environment Programming Guide*
- *Language Environment Programming Reference*.

To create a translation table you must find the code pages for the code sets corresponding to the application requester (on the TPF system) and the application server (on the remote server). These code pages reside either in the data sets for GENXLT support for LE or on the CD-ROM that accompanies the CDRA registry. If the code pages are on the CD-ROM bring them online according to the instructions that follow (see "Creating Translation Tables That Do Not Exist in the LE Data Sets" on page 321). Once online, the procedure for creating the translation table is the same.

When the Translation Table Data Sets are Online

When the data sets that contain the desired translation tables are found in LE GENXLT support, the code page data sets do not need to be loaded from the CDRA registry CD-ROM. The data sets are identified by using the two-letter codes that identifies each code page. This two-letter code is found in the CDRA registry. For instance, the two-letter code, EO, identifies code page 500 and the two-letter code, I1, identifies code page 819. The two-letter codes are joined (source-target) to identify a translation table (EDCUI1EO, where EDCU is a common file name prefix for translation tables).

Having located the data set with the correct translation table, the EDCGNXLT load module creates an object file that can be used on the online TPF system. EDCGNXLT is called using JCL, as shown in the example (in Figure 58).

```
*********************************************************************
*                                                                   *
* LANGUAGE ENVIRONMENT FOR MVS & VM                                 *
*                                                                   *
* EDCGNXLT--- INVOKE THE GENXLT UTILITY                             *
*                                                                   *
* RELEASE LEVEL: vv.rr.mm  (VERSION.RELEASE.MODIFICATION LEVEL)     *
*                                                                   *
*********************************************************************
*                                                                   *
   INFILE=,                    < INPUT DATA SET CONTAINING A SOURCE
                                 TRANSLATION FILE
   REGSIZ='6144K',             < GENXLT REGION SIZE
   OPT=,                       < GENXLT OPTIONS
   OUTFILE=,                   < OUTPUT DATA SET FOR GENERATED OBJECT
   LIBPRFX='PROD.CEE.V1R4M0'   < PREFIX FOR LIBRARY DSN CONTAINING
                                 GENXLT MODULE
*-------------------------------------------------------------
* EDCGNXLT STEP:
* INVOKE EDCGNXLT MODULE TO READ THE SOURCE TRANSLATION FILE
* AND PRODUCE AN OBJECT SUITABLE TO BE LINKED AND LOADED ON TPF.
*-------------------------------------------------------------
EDCGNXLT  EXEC PGM=EDCGNXLT,REGION=&REGSIZ,PARM='&OPT'
STEPLIB   DD DSNAME=&LIBPRFX .SCEERUN,DISP=SHR
SYSIN     DD DSN=&INFILE,DISP=SHR
SYSPUNCH  DD DSN=ACP.DRVE.TEST.OB2(CPG0),DISP=OLD,
             DCB=(BLKSIZE=400,DSORG=PO)
SYSPRINT  DD SYSOUT=*
```

Figure 58. A GENXLT Procedure for Preparing Translation Files

In the previous example, the INFILE data set is an input translation table that correspond to the code pages of the application server of the remote server and the TPF system application requester (EDCUI1EO).

GENXLT support is documented in the *Language Environment Programming Guide*.

The following shows a connection from the TPF system to a server on a RS/6000 system. The TPF system code page is 500 and the RISC code page is 819. The translation table used as input on the INFILE statement in the JCL is found in the *Language Environment Programming Guide*. I1 identifies code page 819 and EO identifies code page 500. So, the input translation table is EDCUI1EO. For the TPF system the standard file name prefix EDCU is dropped and the object module is called I1EO. This prevents conflicts with any name spaces in the Language Environment. Figure 59 on page 319 shows the modified JCL.

```
*********************************************************************
*                                                                   *
* LANGUAGE ENVIRONMENT FOR MVS & VM                                 *
*                                                                   *
* EDCGNXLT--- INVOKE THE GENXLT UTILITY                             *
*                                                                   *
* RELEASE LEVEL: vv.rr.mm  (VERSION.RELEASE.MODIFICATION LEVEL)     *
*                                                                   *
*********************************************************************
*                                                                   *
   INFILE=,                    < INPUT DATA SET CONTAINING A SOURCE
                                 TRANSLATION FILE
   REGSIZ='6144K',             < GENXLT REGION SIZE
   OPT=,                       < GENXLT OPTIONS
   OUTFILE=,                   < OUTPUT DATA SET FOR GENERATED OBJECT
   LIBPRFX='PROD.CEE.V1R4M0'   < PREFIX FOR LIBRARY DSN CONTAINING
                                 GENXLT MODULE
*------------------------------------------------------------
* EDCGNXLT STEP:
* INVOKE EDCGNXLT MODULE TO READ THE SOURCE TRANSLATION FILE
* AND PRODUCE AN OBJECT SUITABLE TO BE LINKED AND LOADED ON TPF.
*------------------------------------------------------------
EDCGNXLT  EXEC PGM=EDCGNXLT,REGION=&REGSIZ,PARM='&OPT'
STEPLIB   DD DSNAME=&LIBPRFX .SCEERUN,DISP=SHR
SYSIN     DD DSN='INPUT.TRANSLATE.TABLE.SCEEGXLT(EDCUI1EO)',DISP=SHR
SYSPUNCH DD DSN='MY.OBJECT.DATA.OB(I1EO)',DISP=OLD,
             DCB=(BLKSIZE=400,DSORG=PO)
SYSPRINT  DD SYSOUT=*
```

Figure 59. A GENXLT Procedure for Preparing Translation Files Filled Out

The object I1EO that contains the translation table is placed in user data set MY.OBJECT.DATA.OB by the EDCGNXLT program.

Packaging Translation Tables

The translation table object files are built into individual DLMs. The names of the DLMs and object files must not match (so that the correct entry point is available). The CPG0 DLM name is reserved. The TPF system uses DLMs named CPG*x* and DLM build scripts named CPG*x*BS. Added DLMs name should be in the user file name space.

The translation table DLM is created using a build script and the CBLD tool, a part of SIP. The build script JCL must refer to the CSTRTD and CENTPT modules, followed by the name of the translation table object module. Figure 60 shows a typical translation table DLM build script.

```
################################################################

DLM CPG1vv            # Include start-up code for DLM

#Object File          Function
#-----------          --------

CENTPT                # return entry point address
I1E0                  # GNXLT translation table 500-819
```

Figure 60. Build Script for a Single-Byte Translation Table Object File

When the CBLD tool reads this build script as input, it produces JCL with an input to the linkage editor as shown in Figure 61 on page 320.

```
//PLKED.SYSIN DD *
 INCLUDE OBJLIB(CSTRTD40)
 INCLUDE OBJLIB(CENTPT)
 INCLUDE OBJLIB(I1E0)
/*
//LKED.SYSLMOD DD DISP=OLD,DSN=ACP.DEVP.TEST.LK(CPG1vv)
```

Figure 61. DLM Build Script JCL for a Single-Byte Translation Table Object File

An additional object module, CHCS, is required when the complex converter is used. The complex converter is a program that handles the conversion between mixed-byte code pages. For example, conversion from Japanese EBCDIC to Japanese ASCII (for example, 939 to 930) requires the use of the complex converter. The complex converter is not used when single-byte code pages (for example, 500 to 819) are translated. Figure 62 shows the complex converter build script.

```
#################################################################

DLM CPG1vv            # Include start-up code for DLM

#Object File          Function
#-----------          --------

CENTPT                # return entry point address
CHCS                  # complex converter
EVEU                  # GNXLT translation table 939-930
```

Figure 62. Build Script for a Mixed-Byte Translation Table Object File

Loading Translation Tables Online

Before loading the translation table DLM to an online TPF system, the name of the DLM and the various kinds of code page identifications are added to two system structures:

1. The CPGS segment contains a table relating the translation table object name and the DLM where the translation table resides. Figure 63 shows a sample translation table with object I1E0 residing in DLM CPG0.

```
table_entry table[table_len] =
{
  {"I1E0","CPG0"},      /*single 819 to single 500*/
  {"EVEU","CPG1"},      /*mixed 939 to mixed 930*/
  {"AAEY","CPG2"},      /*single 850 to 1047 */
  {"AAE0","CPG3"},      /*single 850 to single 500*/
  {"ABEX","CPG5"},      /*mixed 932 to single 1027*/
  {"ABEL","CPG6"},      /*mixed 932 to single 290*/
  {"ABEN","CPG7"},      /*mixed 932 to double 300*/
  {"ACEX","CPG8"},      /*single 897 to single 1027*/
  {"ABEU","CPG9"},      /*mixed 932 to mixed 930*/
  {"0000","0000"},              /*end of table*/
};
```

Figure 63. Table of Translation Table Object Names and DLMs

2. The CSNM segment, an extract of which appears in Figure 64 on page 321, contains various kinds of information that is used to identify a code page.

```
CSNMC CODESET='IBM-500',CODE='EO',CCSID=500,STYLE=S
CSNMC CODESET='IBM-819',CODE='I1',CCSID=838,STYLE=S

CSNMC CODESET='IBM-930',CODE='EU',CCSID=930,STYLE=M,
       SINGLE=290,DOUBLE=300
CSNMC CODESET='IBM-939',CODE='EV',CCSID=5035,STYLE=M,
       SINGLE=1027,DOUBLE=300
```

Figure 64. CSNM Table Showing Code Set Information

> The same information for any code sets that are used in a new translation table must appear in the CSNM table, whether a code set is in the table initially or is added by a user. Required information is available from the CDRA registry and the *Language Environment Programming Guide*.

Once these system structures are updated, the translation table DLMs can be linked and loaded using the TPF loader (TPFLDR) like any other DLM. System operators use the ZSQLD command to add or modify relational database definitions with CCSID and TPFCCSID parameters identifying the new translation tables.

Creating Translation Tables That Do Not Exist in the LE Data Sets

A CD-ROM that contains all possible translation tables is shipped with the character data representation architecture (CDRA) registry. Using the instructions provided in CDRA registry, load a translation table from the CD-ROM. This translation table must be processed into a form that can be read by GENXLT. Figure 65 shows an example of a tool to put the translation table into the correct form.

```
/* */
arg fn ft fm
'pipe < 'fn ft fm ,
'| fblock 1                     ',  /*one char wide */
'| spec number from 0 1.3 1-1 4 ',  /*add record number */
'| spec 1.3 d2x 1 4.1 9         ',  /*convert it to hex */
'| spec 7.2 x2c 1.1 9.1 2       ',  /*convert those to chars */
'| Specs 1-* C2X 1              ',  /*convert it all to text */
'| spec /0x/ 1 1.2 next /0x/ 10 3.2 next /<comment>/ 15  ',
'| > 'fn ' twocolum a           '
```

Figure 65. Sample Tool to Convert CD-ROM Translation Table Data

Once the translation table is online and in a form acceptable to the EDCGNXLT program, the procedure proceeds as though the translation table was found in the data sets for GENXLT support.

Summary of Steps Needed to Create a New Translation Table:

1. Creating the translation table:
   a. Identify the code pages of the server and requestor.
   b. Ensure the Language Environment support is installed on MVS.
   c. Create JCL to call EDCGNXLT to create a translation table.
2. Packaging the translation table:
   a. Name restriction: the DLM name cannot contain an object with same name as the DLM.
   b. Create the build script with the following entries:
      CSTRTD
      CENTPT

I1EO (or the name of the object created by GENXLT).

   c. If this is a complex converter (such as, Japanese EBCDIC to Japanese ASCII) the entries are:
      CSTRTD
      CENTPT
      CHCS
      I1EO.

3. Loading the translation tables to the TPF system:
   a. Update segment CPGS to map I1EO to CPG*x* (or a user DLM)
   b. Check segment CSNM to make sure there is an entry for this code page.
   c. Build and load DLM CSNM.

Installing Additional ISO-C Library Functions

There are several mechanisms at work to create library functions for ISO-C.

Prelinking and Linking

Function libraries are nothing more than previously compiled code that can be used by newly compiled code for some purpose. Frequently used ISO-C code can be collected into common library routines available to all ISO-C programs. System programmers may create site dependent versions of C input/output functions (like `gets` and `puts`). Similarly, extensions to ISO-C libraries can be used to enforce common programming techniques and support specific kinds of data. These libraries can be used with the libraries shipped by IBM to enhance the C programming environment.

The process of making code in libraries available to user programs is called *linkage*. It is performed by a *linkage editor*. In ISO-C, there is prelinkage before linkage.

Functions that are called in the user program are represented in the stub library. For example, the user can call for the `malloc` function in their program. There is a stub for `malloc` which consists of a library lookup for the code that actually implements the action of `malloc`.

The library vector (LIBVEC) consists of slots for addresses that show the location of each function in the library. The linkage editor uses the library vector and the composite object file to create the load module containing the machine code and relocation information for the user's program. The load module ultimately is loaded and run on TPF. See Figure 66 on page 323 to see how the offline support of ISO-C relates to the online support.

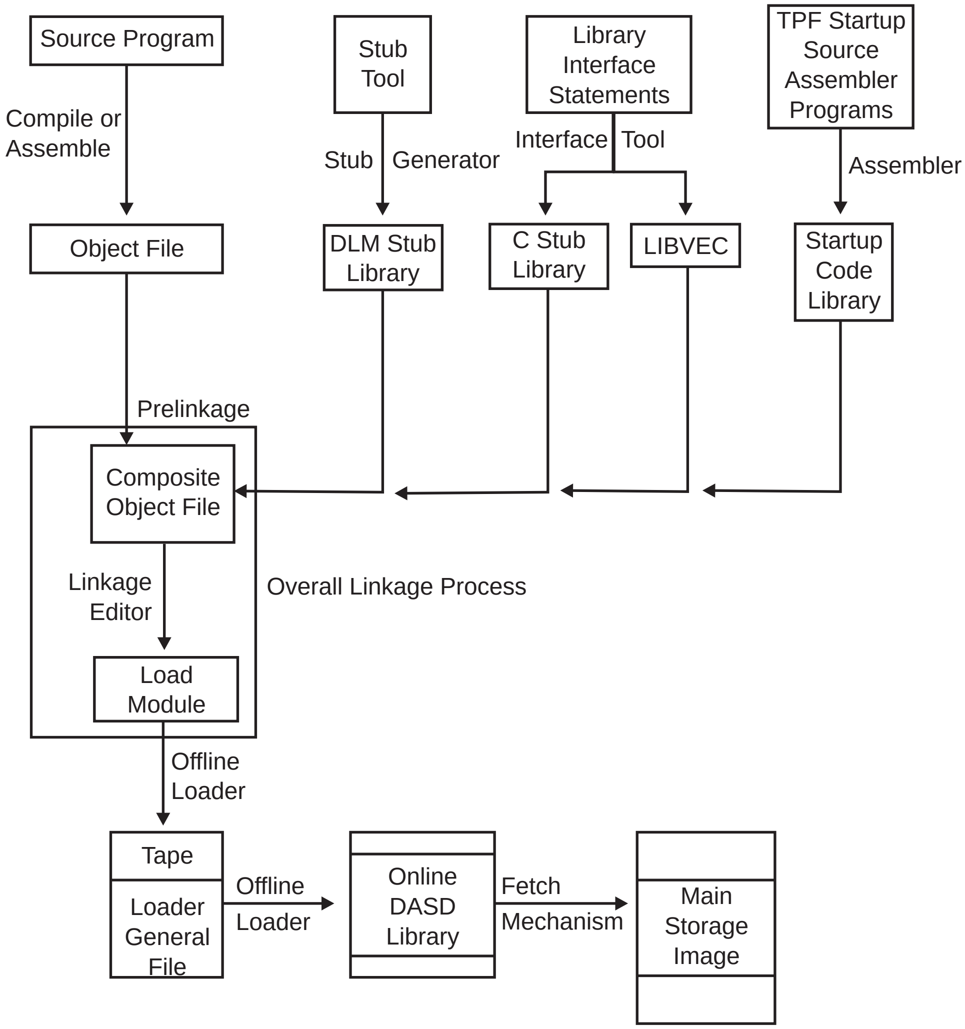


Figure 66. ISO-C Compile, Link, and Load Process

Library Interface Tool

This section describes the library interface tool.

Purpose

The library interface tool produces a library transfer vector (LIBVEC) and the library call linkage stubs. Figure 67 on page 324 shows the relationships found in the tool. To do this, the tool reads a file defining the library ordinal numbers and the functions contained in the libraries. These files are called *library interface scripts*. The library interface scripts define the libraries using @libid and @libfun statements.

Note

You do not need the library interface tool for dynamic link libraries (DLLs).

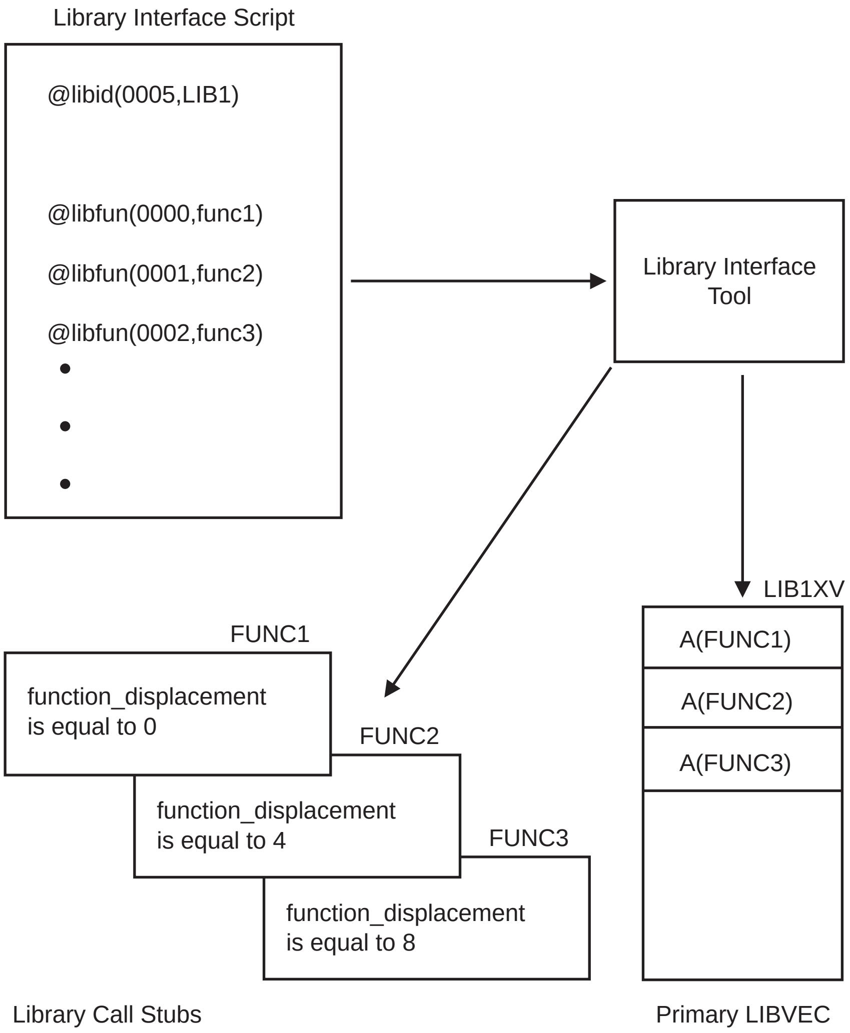


Figure 67. Output of the Library Interface Tool

Library ordinals are extremely important because they identify the library in the online system (using the array of library addresses). There are 1024 library ordinals available and 1024 functions can appear in each library. So, you can refer to more than 1 million functions throughout all the possible libraries. Of the the library ordinals, IBM reserves ordinals in the 0–199 range for its own use and the remainder (200–1023) for users.

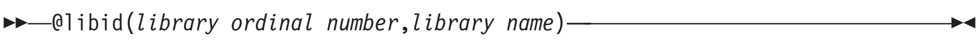
The first non-comment line in a build script defines the library ordinal and library name. The remaining non-comment lines specify the ordinal numbers and names for the functions in the library.

Requirements and Restrictions

- The ordinal numbers for the libraries and the functions specified in the library interface scripts must correspond exactly to the ordinals for the libraries and functions loaded online.
- Each function defined in the library interface script must also be defined in one of the object files included in the library, or the results of calling the function are undefined.

- Some common rules govern interface tool statements. Each statement is restricted to a single line. Comments begin with the number sign (#) and continue for the remainder of the input line. Blank lines are ignored.
- Refer to *TPF General Macros*, *TPF System Macros*, or *TPF Operations* for information on how to read syntax diagrams that represent the statement formats.

Format for the @libid Statement



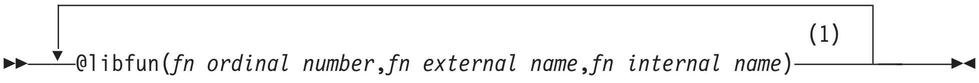
library ordinal number
the decimal ordinal number of the library

library name
the name of the library

Additional Information

- Only 1 @libid statement can be used in each library interface script
- Every library must have a unique ordinal number. Each library should have the same ordinal on every processor where it is loaded. This is especially important when using libraries or code imported from another installation.
- The name is 4 characters long and must be unique.

Format for the @libfun Statement


@libfun(*fn ordinal number*,*fn external name*,*fn internal name*)
(1)

Notes:

1 Each repetition takes place on a new line.

fn ordinal number
the decimal ordinal number of the function. The function ordinal is the number of the function in the library. The position in the file of the @libfun statement specifying a function has no effect on the ordinal of the function.

fn external name
The external name is the name of the function to be used by callers (users of the library). The internal_name is the name of the function in the library.

fn internal name
The internal name is optional and defaults to the external_name if it is not used.

Additional Information

- Only 1 @libfun statement should appear for each function being defined.

- For each @libfun statement you must also define a library function within the library load module. The name of the function is the same as the *internal* name defined by the `@libfun` statement (if *fn internal name* is not specified, then the internal name is the same as *fn external name*). If no such function is defined, a call stub for the function will still be generated, and the results of calling the function will be indeterminate. During development, before the library is fully implemented, coding an empty function that just returns, or one that only takes a snap dump, for each unimplemented function will ensure that the library works in a predicatable manner.

Example of a Library Interface Tool Script

This script creates a small ISO-C library. There are several points to notice:

- The name of the library, ASHB, appears in the @libid statement. The ordinal number of the ASHB library is 13. Notice that the line immediately following the @libid statement is blank.
- The order that the functions appear in the library depends solely on the ordinal specified. The order they appear in the list has no bearing on the order of functions in the library. It is not required that the ordinals be in increasing order.
- Function 8, good1, was commented out. It does not appear in the finished library.
- The cursr function is defined to appear at the end of the library generated. The programmer who created the list ensured that it would appear as the last function in the library by making its ordinal 1023. Where cursr appears in the input list has no effect.
- The die function has an internal name as well as an external name. The internal name (_TPFABRT) is what the members of the library are called by the library, and the external name (die) is used by callers.

```
########################################################################

@libid(0013,ASHB)                      # Ashby's Gems

@libfun(0000,mysort)                   # Called by payroll program
@libfun(0001,die,_TPFABRT)             # stop it, please
@libfun(0003,qsort1)                   # quicker
@libfun(0005,qsort2)                   # maybe
@libfun(0007,token)                    # get next word

@libfun(0002,hello)
@libfun(0004,mydate)                   # provides sensible date data
@libfun(0006,myssm)
#####libfun(0008,good1)                # Use another version

@libfun(1023,cursr)                    # last function in library
```

Running the Tool

The tool runs as an offline job. It uses several data sets as specified in the data sets table on page Table 26.

Table 26. File Specifications for the Library Interface Tool

DD Name	DCB Characteristics	Purpose

SYSIN	LRECL=80,
	RECFM=F or
	RECFM=FB
		Input file of library interface statements - the interface script

SYSPRINT		Output file for tool messages

Table 26. File Specifications for the Library Interface Tool (continued)

DD Name	DCB Characteristics	Purpose

SYSXV	LRECL=80,
	RECFM=F or
	RECFM=FB
		Output object file for the library transfer vector (LIBVEC)

SYSCLS	LRECL=80,
	RECFM=F or
	RECFM=FB
		Output object partitioned data set (PDS) containing the library call linkage stubs.

There are 2 run-time parameters: MSGS and SOURCE. They regulate the information provided to SYSPRINT.

MSGS
lists only library interface tool messages. This is the default.

SOURCE
lists the input statements interspersed with any messages from the library tool.

The JCL necessary to run the interface script is straight forward. The program name, LIBIvv, is requested to run from the ACP.LINK.RELvv library. According to our previous example the interface statements are found in a file called ASHBXVvv in the ACP.CSRCE.RELvv dataset. SYSPRINT is to the standard output file, SYSOUT. The library vector (LIBVEC) is put in the LIB1XVvv member of ACP.CSRCE.RELvv, overriding any existing version as LIBI proceeds. The stub library is put in the ACP.CLIB.RELvv file. Any error messages displayed are documented in the Offline section of *TPF Messages, Volume 1* and *TPF Messages, Volume 2*.

```
//LIBINTFC EXEC PGM=LIBIvv,REGION=4M
//STEPLIB  DD DSN=ACP.LINK.RELvv.BSS,DISP=SHR
//         DD DSN=LE.V1R3M0.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYSXV    DD DSN=ACP.OBJ.RELvv.BSS(ASHBXVvv),DISP=OLD
//SYSCLS   DD DSN=ACP.CLIB.RELvv.BSS,DISP=OLD
//SYSIN    DD  DSN=ACP.CSRCE.RT.RELvv(ASHBXVvv),DISP=SHR
//SYSPRINT DD SYSOUT=A
/*
SOURCE
MSGS
/*
```

Return codes indicate if the tool successfully built the library vector and stub library.

Code	Meaning
0	indicates successful completion and informational messages
4	indicates unsuccessful completion and one or more warning messages as the highest severity
8	indicates unsuccessful completion and 1 or more error messages as the highest severity
16	indicates unsuccessful completion.

DLM Call Stub Generator

This section describes the DLM call stub generator.

Purpose

The DLM call stub generator adds stubs to an object library of call stubs. Any entry point function called by an ISO-C load module must have a stub produced by the generator. The stubs are statically linked to an ISO-C program using the autocall mechanism during prelinking. The program name and PAT displacement is generated as a weak TPF VCON in the stub object module. The weak VCON is resolved when the offline loader runs.

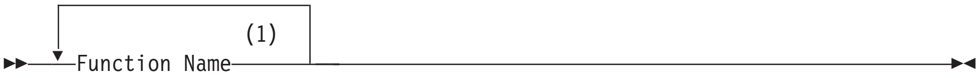
Note

All load modules need the DLM call stub generator to generate the stubs for other E-type programs external to themselves that are called with TPF Enter/Back services.

Requirements and Restrictions

- The input file is a list of 4-character program names, 1 per line.
- Comments are specified by the number sign (#) and are similar to the comments in the library interface tool.
- Stubs can be produced for allocated or unallocated programs. The names are not checked against the program allocator list (PAL).
- Blank lines are ignored.

Format for a DLM Stub Generator Statement



Notes:

1 Each repetition takes place on a new line.

Function Name	Name of a function found in any DLM.

Running the Generator

Table 27 lists the data sets required by the stub generator.

Table 27. File Specifications for the DLM Call Stub Generator

DD Name	DCB Characteristics	Purpose

SYSIN	LRECL=80,
	RECFM=F or
	RECFM=FB
		Input file of DLM call stub generator statements - the generator script.

SYSPRINT		Output file for generator messages.

STUBS	LRECL=80,
	RECFM=F or
	RECFM=FB
		Output object partitioned dataset (PDS) generated.

The JCL needed to run the stub generator is quite simple. The generator (CSTUBGvv) is set up to run using the version on the ACP.LINK.RELvv library. The listing goes to SYSPRINT, which is set to standard output. The object file file containing the stubs is written through DD name STUB in the PDS ACP.STUB.RELvv. The input to the generator appears right after the SYSIN DD *.

There are 3 comment lines followed by 3 lines indicating routines to put into the stub object file. Comments follow each routine showing where the calling routines are.

```
//CSTUBGEN EXEC PGM=STUBvv,REGION=4M
//STEPLIB  DD DSN=ACP.LINK.RELvv.BSS,DISP=SHR
//         DD DSN=LE.V1R3M0.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//STUB     DD DSN=ACP.STUB.RELxx.BSS,DISP=OLD
//SYSPRINT DD SYSOUT=A
//SYSIN    DD  *
#
# Generate 3 stub object modules
#
PGM1            # called by DLM1
PGM2            # called by DLM2
UIIO            # called by DLM1 and DLM2
/*
```

Return codes indicate if the generator successfully built the stub object file. A stub is generated for each program name in the input file. If an error occurs or a problem prevents stub generation, an error message is displayed in SYSPRINT and a nonzero return code is set. Each error message contains the line number of the corresponding input line. The return codes are:

0	indicates successful completion and all stub object modules generated
4	indicates a warning that there were no program names provided in the input file, so no stubs were generated.
8	indicates an error that prevented 1 or more stub object modules from being generated.
16	indicates that the error message file could not be opened.

ISO-C Load Module Build Tool (CBLD)

This section describes the ISO-C load module build tool.

Purpose

The C load module build tool (CBLD) creates the job control language (JCL) needed to make files from compiled or assembled object files. Input to the C load module build tool consists of build script input records, collectively called a *script*, and comments.

The C load module build tool is called by SIP to generate the JCL needed to prelink and link-edit run-time (nondynamic) libraries, DLMs, or dynamic link libraries (DLLs). It can be used as a front-end utility to assist application programmers.

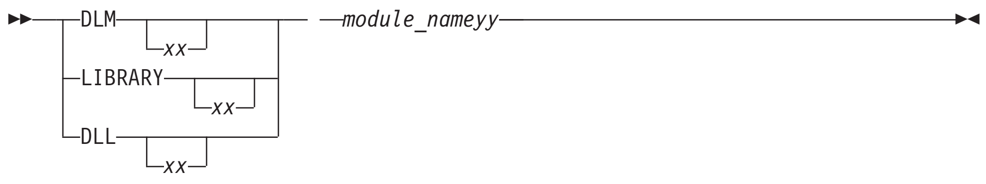
Requirements and Restrictions

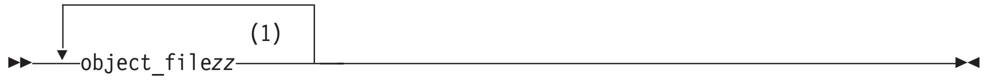
- The first build script input record must specify DLM, LIBRARY, or DLL, followed by the load module name and version. Load module names are 4 characters long and are immediately followed by a 2-character version. Object file names and versions follow, 1 per line.
- The requirement for an object file is to be 4 to 6 characters, followed by a 2-character version. You can code the name anywhere in columns 1—72; the convention is to start the name in column 1.

  Note: See *TPF Programming Standards* for more information about naming conventions.
- Comments begin with a number sign (#) and continue to the end of the source record. The beginning of a comment can occur anywhere in a library interface instruction. There are no special continuation characters to join 2 lines.
- The build tool adds an INCLUDE OBJ line in the output JCL file. If no characters are in columns 1–72, the line is considered blank and ignored. This allows for library systems that put sequence numbers in columns 73–80. There are no continuations. The build tool does not provide a warning message.

  Note: The data set itself can have a RECFM of V and an LRECL greater than 73.
- If an `@IMPORTDS` statement is in the build script, CBLD will include data sets in the output JCL file for definition side-decks imported by the DLL.
- No check is made for duplicate object file names in the output JCL.

Format for a Load Module Build Tool Statement





Notes:

1 Each repetition takes place on a new line.

module_name
Name of the load module being created.

xx yy zz
Two-digit, alphanumeric version codes.

When DLM, LIBRARY, or DLL has a version code, the version appears on the DLM, library, or DLL startup code (CSTRTD*xx*, CSTRTL*xx*, or CSTDLL*xx*, respectively). If DLM, LIBRARY, or DLL does not have a version code supplied, version 40 is used as the default. Similarly, if the version code supplied is not valid, version code 40 is used.

Sample Load Module Build Scripts

The following shows an example of a build script for a dynamic load module (DLM). The load module in this example is a non-DLL application because there are no `@IMPORTDS` statements.

```
######################################################################
#                                                                    #
#  SCRIPT NAME..... CDM0BS                                           #
#                                                                    #
   ⋮
#                                                                    #
######################################################################

DLM CDM0LB  # Include startup code for DLM

#Object File  Function                           Source Language
#-----------  --------                           ---------------

CDMAINLB      # ZDMAP command                    C
              # mainline routine
CDMPRSLB      # zdmap_parse                      C
CDMHLP40      # zdmap_help                       C
CDMDSPF1      # zdmap_display                    C
CDMER140      # zdmap_parse_error_handler; error C
              # messages in 1000 range
CDMER240      # zdmap_retrieve_error_handler;    C
              # error messages in 2000 range
CDMER340      # zdmap_process_error_handler;     C
              # error messages in 3000 range
```

The load module that results from this sample script is CDM0LB, where LB is the version code.

Note: With link map support, if the object file name plus the version code is greater than 8 characters, an error message will be written to the message file and the INCLUDE card will not be written to the generated JCL deck.

The following shows an example of a build script for a DLL.

```
######################################################################
#                                                                    #
#  SCRIPT NAME..... QZZ2BS                                           #
#                                                                    #
   ⋮
#                                                                    #
######################################################################

DLL QZZ2RX  # Include startup code for DLL

@IMPORTDS CPP140  # Include a definition side-deck
@IMPORTDS DLL3RK  # Include a definition side-deck

#Object File  Function                           Source Language
#-----------  --------                           ---------------

QZZ2A41       # QZZ2A function                   C++
QZZ2B41       # QZZ2B function                   C
```

Note: With the `DLL` keyword, CSTDLL startup code is used. The `@IMPORTDS` statements indicate the definition side-decks that this DLL imports from. `@IMPORTDS` statements must follow the DLL statement and precede the list of object files to be included.

The following shows an example of a build script for a DLM that is a DLL application.

```
#######################################################################
#                                                                     #
#  SCRIPT NAME..... QZZ1BS                                            #
#                                                                     #
   ⋮
#                                                                     #
#######################################################################

DLM QZZ1RX  # Include startup code for DLM (DLL application)

@IMPORTDS QCCC41  # Include a definition side-deck
@IMPORTDS QDDDRX  # Include a definition side-deck

#Object File  Function                           Source Language
#-----------  --------                           --------------

QZZ1A41       # QZZ1A function                   C
QZZ1B41       # QZZ1B function                   C++
```

Note: DLL applications use the `DLM` keyword in the build script. CSTRTD startup code is used. The `@IMPORTDS` statements indicate the definition side-decks that this DLL application (DLM) imports from. `@IMPORTDS` statements must follow the DLM statement and precede the list of object files to be included.

Running the Build Tool

The C load module build tool (CBLD) requires 3 data sets, as described in Table 28.

Table 28. C Load Module Build Tool Data Sets

C Definition	Purpose
stdin	Input file containing a C load module build script. The user specifies the name of a file, called the Load Module Build Script.
stdout	Output file containing JCL to link the object files. The user specifies the name of a file used to write the JCL to link the load modules. This file can be modified before it is run on MVS.
stderr	Output file for C Load Module Build Tool messages. The user specifies the name of a file used to write messages, including error messages. Read this file before the JCL is sent to MVS for execution.

The JCL needed to run the build tool is quite simple.

```
//CLMBUILD EXEC PGM=CBLDxx,REGION=4M
//STEPLIB  DD  DSN=ACP.LINK.INTGvv.NBS,DISP=SHR
//         DD  DSN=LE.V1R3M0.SCEERUN,DISP=SHR
//SYSUDUMP DD  DUMMY
//SYSABEND DD  DUMMY
```

```
//SYSPRINT DD  DSN=ACP.LK.RELxx(myprgvv),DISP=(NEW,PASS),UNIT=SYSDA,
//         DCB=(BLKSIZE=400,RECFM=FB,LRECL=80)
//SYSIN    DD  DSN=ACP.SRCE.OL.INTGvv(myprgvv),DISP=SHR
/*
```

where

- ACP.LINK.RELxx contains the CLBDxx program to be executed.
- ACP.OBJ.RELxx(myprgvv) contains the input file or load module build script.
  SYSIN records can also be defined in the input stream rather than in a data set.
- ACP.LK.RELxx(myprgvv) will contain the output JCL.
- CBLDxx can have the NOJCL parameter added. Adding it looks like:

  ```
  //CBMBLD  EXEC PGM=CBLDxx,PARM='NOJCL',REGION=4M
  ```

  This parameter suppresses the generation of the JCL decks for prelinking and linking, and only generates the list of INCLUDE files. There is no JCL parameter. If NOJCL is not specific, the JCL decks are generated by default.

Return codes indicate if the generator successfully built the JCL object file.

Code	**Meaning**
2	Successful completion with only informational messages.
4	One or more warning messages were generated, but no error or fatal messages.
8	One or more error messages were generated, but no fatal message.
16	The tool ended with a fatal message.

When the build tool completes successfully, a JCL deck has been generated that will link the object library or DLM specified. The linkage editor, EDCPL, takes the object files specified (in this case using SYSIN) and produces a linked load module in the `ACP.LK.RELvv` data set.

For link map support, the generated JCL deck has extra data. For each included object file (excluding the startup code), the following three cards are added before each INCLUDE card:

1. An ESD card for a symbol named @@LM*nnnn*, where *nnnn* is a 4-digit value beginning with 0001.
2. A TXT card, which contains 8 bytes of character data. The first 6 bytes are for the name of the object file and the last 2 bytes contain the version code.

   Note: If the object file name is less than 6 characters, the name is blank-padded on the right. The version code is always in bytes 7 and 8.
3. An END card.

With the addition of these three cards, the prelinker and linkage editor create an 8-byte CSECT before each object file. The symbol named in the ESD card is found by the offline loader so that the address of each 8-byte CSECT can be placed in the link map.

After the INCLUDE card for the last object file, the following cards are added by CBLD to build a final CSECT:

1. An ESD card for a symbol named @@LM*nnnn*, where *nnnn* is a 4-digit value that is one digit higher than the previous value used.

2. A TXT card, which contains the following 16 bytes of eye-catcher character data: 'END_OF_LAST_OBJ '
3. An END card.

This CSECT is used to mark the end of the last object file so that its size can be determined.

The sample JCL that is generated for CDM0LB used a DLM specification on input. This can be seen by the inclusion of the CSTRTD object file (the DLM startup code) at the beginning of the list of object files to include.

The ESD, TXT, and END cards generated by CBLD contain some unprintable characters that cannot be seen in the following examples.

```
//$CDM0BS1 JOB ...
//PRELINK EXEC EDCPL,COND.LKED=(0,NE),
// LPARM='AMODE=31,RMODE=ANY,LIST,XREF'
//PLKED.SYSLIB DD DSN=ACP.CLIB.RELxx,DISP=OLD
//             DD DSN=ACP.STUB.RELxx,DISP=OLD
//PLKED.OBJLIB DD DSN=ACP.OBJ.RELxx,DISP=SHR
//PLKED.SYSIN DD *
 INCLUDE OBJLIB(CSTRTD40)
 ESD            @@LM0001
 TXT            CDMAINLB
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMAINLB)
 ESD            @@LM0002
 TXT            CDMPRSLB
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMPRSLB)
 ESD            @@LM0003
 TXT            CDMHLP40
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMHLP40)
 ESD            @@LM0004
 TXT            CDMDSPF1
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMDSPF1)
 ESD            @@LM0005
 TXT            CDMER140
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMER140)
 ESD            @@LM0006
 TXT            CDMER240
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMER240)
 ESD            @@LM0007
 TXT            CDMER340
 END                            1569623400 010195215
 INCLUDE OBJLIB(CDMER340)
 ESD            @@LM0008
 TXT            END_OF_LAST_OBJ
 END                            1569623400 010195215
/*
//LKED.SYSLMOD DD DSN=ACP.LK.RELxx(CDM0LB),DISP=OLD
//
```

As mentioned previously, object file names that are less than 6 characters are right-padded with blanks. Using QZZ0 as an example, the build script for C load module QZZ0 is as follows:

```
DLM QZZ0LD  # Include startup code for DLM

#Object File  Function                           Source Language
#-----------  --------                           ---------------
```

```
QZZ0LD        # QZZ0 function                    C
QZADDLC       # QZADD function                   C
```

After running the C load module build tool, the output is as follows:

```
//PLKED.SYSIN DD *
 INCLUDE OBJLIB(CSTRTD40)
 ESD           @@LM0001
 TXT           QZZ0  LD
 END                            1569623400 010195215
 INCLUDE OBJLIB(QZZ0LD)
 ESD           @@LM0002
 TXT           QZZ1A LC
 END                            1569623400 010195215
 INCLUDE OBJLIB(QZZ1ALC)
 ESD           @@LM0003
 TXT           END_OF_LAST_OBJ
 END                            1569623400 010195215
/*
```

Note: The `COND` parameter on the `EXEC` card means “execute the link-edit step only if the prelink step returns zero.” For online TPF ISO-C programs, all external references should be resolved by the prelinker and, therefore, a nonzero return code from the prelink step indicates an error in the build process. For regular MVS C programs (including offline TPF C programs), an RC=4 from the prelink step is normal because references to library functions are only resolved during the link-edit step.

CBLD Support for Dynamic Link Libraries (DLLs)

Both the previous DLL and DLL application build script examples show the use of the `@IMPORTDS` keyword. CBLD takes the load module name that follows this keyword and adds a definition side-deck `INCLUDE` statement to the list of `INCLUDE` statements in the JCL output deck. For a DLL only, the JCL will also contain the `ACP.IMPORTS.RELvv` definition side-deck `SYSDEFSD` data set for exported functions and variables. The following examples show the different JCL output decks generated by CBLD for a DLL and a DLL application:

- For a DLL:

```
//$QZZ2BS0 JOB ...
//PRELINK EXEC EDCPL,COND.LKED=(0,NE),
// LPARM='AMODE=31,RMODE=ANY,LIST,XREF,MAP',
// PPARM='DLLNAME(QZZ2)'
//PLKED.SYSLIB DD DSN=ACP.CLIB.RELxx,DISP=OLD
//             DD DSN=ACP.STUB.RELxx,DISP=OLD
//PLKED.OBJLIB DD DSN=ACP.OBJ.RELxx,DISP=SHR
//*
//PLKED.SYSDEFSD DD DISP=SHR,DSN=ACP.IMPORTS.RELxx(QZZ2RX)
//*
//PLKED.DSD DD DISP=SHR,DSN=ACP.IMPORTS.RELxx
//PLKED.SYSIN DD *
 INCLUDE OBJLIB(CSTDLL40)
 INCLUDE DSD(DLL240)
 INCLUDE DSD(DLL3RK)
 ESD           @@LM0001
 TXT           QZZ2A 41
 END                            1569623400 010195215
 INCLUDE OBJLIB(QZZ2A41)
 ESD           @@LM0002
 TXT           QZZ2B 41
 END                            1569623400 010195215
 INCLUDE OBJLIB(QZZ2B41)
 ESD           @@LM0003
 TXT           END_OF_LAST_OBJ
 END                           1569623400 010195215
/*
//LKED.SYSLMOD DD DISP=OLD,DSN=ACP.LK.RELxx(QZZ2RX)
//
```

- For a DLL application:

```
//$QZZ1BS5 JOB ...
//PRELINK EXEC EDCPL,COND.LKED=(0,NE),
// LPARM='AMODE=31,RMODE=ANY,LIST,XREF'
//PLKED.SYSLIB DD DSN=ACP.CLIB.RELxx,DISP=OLD
//             DD DSN=ACP.STUB.RELxx,DISP=OLD
//PLKED.OBJLIB DD DSN=ACP.OBJ.RELxx,DISP=SHR
//PLKED.DSD DD DISP=SHR,DSN=ACP.IMPORTS.RELxx
//PLKED.SYSIN DD *
 INCLUDE OBJLIB(CSTRTD40)
 INCLUDE DSD(QCCC41)
 INCLUDE DSD(QDDDRX)
 ESD           @@LM0001
 TXT           QZZ1A 41
 END                           1569623400 010195215
 INCLUDE OBJLIB(QZZ1A41)
 ESD           @@LM0002
 TXT           QZZ1B 41
 END                           1569623400 010195215
 INCLUDE OBJLIB(QZZ1B41)
 ESD           @@LM0003
 TXT           END_OF_LAST_OBJ
 END                           1569623400 010195215
/*
//LKED.SYSLMOD DD DISP=OLD,DSN=ACP.LK.RELxx(QZZ1RX)
//
```

See the programming guide for the IBM C/C++ compiler used by your installation for more information about DLLs and definition side-decks. See *TPF System Generation* for more information about the `ACP.IMPORTS.RELvv` definition side-deck data set.

Rearrange TXT (REATXT) Tool: Sample Code Only

For link map support, the REATXT tool (sample code only) performs the extra processing needed when the C source file has been compiled with the NORENT option and the VM linkage editor (LKED) is used. The code and static data CSECTs for each object file are out of order, making it difficult to find the code that matches up with a compiled listing. The REATXT tool rearranges the CSECTs to match the order produced by the MVS linkage editor between the prelink and link-edit steps.

Note: If you do all the compiles with the RENT option and use the VM linkage editor (VM LKED), you do not need to use the REATXT tool.

Table 29 shows you the difference between compiling with the NORENT option while using the MVS linkage editor versus using the VM linkage editor (VM LKED).

Table 29. MVS and VM Linkage Editor Comparison

MVS	VM LKED + NORENT
CSTRTD	CSTRTD
Address of object file 1 Code CSECT for object file 1 Static CSECTs for object file 1	Address of object file 1 Static CSECTs for object file 1 Code CSECT for object file 1

Table 29. MVS and VM Linkage Editor Comparison (continued)

MVS	**VM LKED + NORENT**
Address of object file 2 Code CSECT for object file 2 Static CSECTs for object file 2	Address of object file 2 Static CSECTs for object file 2 Code CSECT for object file 2
ESDs are placed in ascending numeric order.	ESDs are placed in ascending numeric order.
TXTs are in the same ascending numeric order as ESDs.	TXTs are left in the same order from the compile.

Because the length of these static variable CSECTs is not a fixed length, it is helpful to rearrange the CSECTs to match the order produced by the MVS linkage editor between the prelink and link-edit steps.

See the prolog of the REATXT source code for more information about using the REATXT tool.

Sample JCL for Generating ISO-C Offline Tools

The following sample JCL shows running the library generator, the stub generator, and the linkage editor in a single job stream. These jobs were generated by SIP and can be identified by their job names (SIPLxx).

```
//*****************************************************
//*
//*  Library Interface Tool
//*  - Build the CTALXV library
//*
//*****************************************************
//SIPL51 JOB  (82F91,7323E),'SIP ACP  ',
//   MSGLEVEL=1,CLASS=F,
//   MSGCLASS=A,TIME=100
/*ROUTE PRINT TPFVM1(EDDYE)
/*ROUTE PUNCH TPFVM1(EDDYE)
//L511B EXEC PGM=LIBI40,REGION=4M
//STEPLIB  DD DSN=ACP.LINK.RELxx.BSS,DISP=SHR
//         DD DSN=LE.V1R3M0.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYSXV    DD DSN=ACP.OBJ.RELxx.BSS(CTALXV40),DISP=OLD
//SYSCLS   DD DSN=ACP.CLIB.RELxx.BSS,DISP=OLD
//SYSIN    DD  DSN=ACP.CSRCE.RT.RELxx(CTALXV40),DISP=SHR
//SYSPRINT DD SYSOUT=A
//
//*****************************************************
//*
//*  DLM STUB GENERATOR JCL
//*    - build a stub for CYYM
//*
//*****************************************************
//SIPL52 JOB  (82F91,7323E),'SIP ACP  ',
//   MSGLEVEL=1,CLASS=F,
//   MSGCLASS=A,TIME=100
/*ROUTE PRINT TPFVM1(EDDYE)
/*ROUTE PUNCH TPFVM1(EDDYE)
//L521A EXEC PGM=STUB40,REGION=4M
//STEPLIB  DD DSN=ACP.LINK.RELxx.BSS,DISP=SHR
//         DD DSN=LE.V1R3M0.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//STUB     DD DSN=ACP.STUB.RELxx.BSS,DISP=OLD
//SYSPRINT DD SYSOUT=A
//SYSIN    DD  *
CYYM
```

```
/*
//*****************************************************
//*
//*  BUILD TOOL JCL
//*   - Build the JCL and INCLUDE decks for CTAL
//*     Note &&INCDECK (NEW,PASS)
//*
//*****************************************************
//
//SIPL53 JOB  (82F91,7323E),'SIP ACP  ',
//   MSGLEVEL=1,CLASS=F,
//   MSGCLASS=A,TIME=100
/*ROUTE PRINT TPFVM1(EDDYE)
/*ROUTE PUNCH TPFVM1(EDDYE)
//L53A EXEC PGM=CBLD40,REGION=4M,PARM='NOJCL'
//STEPLIB  DD DSN=ACP.LINK.RELxx.BSS,DISP=SHR
//         DD DSN=LE.V1R3M0.SEDCLINK,DISP=SHR
//         DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYSPRINT DD  DSN=&&INCDECK,DISP=(NEW,PASS),UNIT=SYSDA,
//         DCB=(BLKSIZE=400,RECFM=FB,LRECL=80)
//SYSIN    DD  DSN=ACP.CSRCE.RT.RELxx(CTAL40),DISP=SHR
//*****************************************************
//*
//*  PRELINK/LINK JCL
//*   - Link CTAL using the EDCPL proc
//*
//*****************************************************
//L53B EXEC EDCPL,COND.LKED=(0,NE),
// LPARM='AMODE=31,RMODE=ANY,LIST,XREF'
//PLKED.SYSLIB  DD DSN=ACP.CLIB.RELxx.BSS,DISP=SHR
//           DD DSN=ACP.STUB.RELxx.BSS,DISP=SHR
//PLKED.OBJLIB  DD DSN=ACP.OBJ.RELxx.BSS,
//           DISP=SHR
//PLKED.SYSIN DD DSN=&&INCDECK,DISP=(OLD,DELETE)
//LKED.SYSLIN  DD DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//LKED.SYSLMOD DD DSN=ACP.LINK.RELxx.BSS(CTAL40),DISP=OLD
//
```

Installing Additional IBM TARGET(TPF) C Library Functions

Periodically, application programmers can want to collect frequently used C code into a common library routine available to all application programs. The following section describes the procedure for doing this.

1. Choose a TPF program segment to contain the library function source code. Prepare the C source code.
2. Add a CLIBFUN macro call to segment C000 for the new function.

   The format of the CLIBFUN macro is:

   ```
   CLIBFUN cfun,tpfname,TYPE=,VERSION=,FEATURE=
   ```

   where:

   cfun
   is the name of the C library function

   tpfname
   is the TPF program segment that will contain the C library function

   TYPE
   is the TPF segment type for the SPPBLD macro. Valid types are:

RT	TPF system assembler code
UR	user assembler code

CRT	TPF system C code
CUR	user C code
OCO	object code only.

VERSION
is the version code on the TPF segment name. The default is 40.

FEATURE
is the name of a TPF feature, such as TPFAR. Functions that belong to optional features need to code this parameter. (This parameter is optional for functions of required features.) The feature name is defined in SYGLB and set by SYSET.

Note: If the feature is not active, the CLIBFUN macro call for that set of functions is considered a dummy call.

cfun and **tpfname** are positional, required parameters. **TYPE** and **VERSION** are keyword parameters, and **VERSION** is optional.

The purpose of segment C000 is to provide a single point of update whenever new library functions are created. C000 contains a CLIBFUN macro call for every library function in the system. When C000 is assembled, 4 sets of objects are automatically produced in the form of *comment cards* in the object decks:

a. A set of `#pragma map` and `#pragma linkage` statements for the C compiler. Remove these `#pragma map` and `#pragma linkage` statements and put them in the `tpflink.h` header file. This header file contains all of the information the compiler needs to provide linkage.
b. A set of allocation statements for the system allocator. Remove these statements and add them to your allocator input deck.
c. A set of SPPBLD statements for the TPF program list. The SPPBLD macro is used to build dependency tables that determine which programs will or will not be included in the system. Extract these SPPBLD cards and add them to SPPGML.
d. A set of LOADER CALL cards for the system loaders. These cards can be extracted and added to the appropriate load deck.

Note: To avoid having to recompile all existing C application programs, add CLIBFUN calls for new library functions to the END of the list in C000. The order in which the CLIBFUN calls appear directly affects the calling linkage produced by the compiler. If the sequence of the CLIBFUN calls is disturbed, you must recompile all application programs that refer to library functions that have moved to ensure that the compiler resolves the linkage to each library function correctly.

You can also code the following special form,

```
CLIBFUN DUMMY,tpfname
```

which will reserve an entry in the quick enter directory for future use. No comment cards will be generated for this item, but it will serve as a placeholder so that the sequence of CLIBFUN calls is not disturbed when the item is later converted to a normal CLIBFUN call. If any program attempts to call a library function using an index number that corresponds to a *dummy slot*, segment C246 will be activated and cause the ECB to exit with system error.

3. Assemble C000 and use a text editor to extract the cards needed from the object deck, as described in Step 2.

   You can want to extract only the allocator statements, SPPBLD, and LOADER CALL cards that are actually required for the new library function. However, be sure to replace the entire set of `#pragma` statements in `tpflink.h` with those generated by assembling C000 as the new set of `#pragma map` and `#pragma linkage` statements are interrelated.
4. Prepare to run SIP.

   Check if an update to the APSIZxx parameter of the CORREQ macro is required to accommodate the additional code. This can be determined by examining the sum of accumulated code size that appears in the object decks (and in the listings) when C000 is assembled.
5. Run SIP Stages I and II. The resulting TPF system will contain the new library function.
6. Be sure to compile any application programs that call the new function against the updated version of `tpflink.h`. This will ensure that the library function linkage is resolved correctly by the compiler.

Removing TARGET(TPF) Library Functions

When you want to remove a TARGET(TPF) C library function follow these steps:

1. Replace the CLIBFUN macro call in segment C000 that corresponds to the library function with a CLIBFUN DUMMY call, and reassemble C000.

   This will preserve the sequence of CLIBFUN calls so that index numbers into the quick enter directory are not disturbed and recompilation of the C application programs will not be required. This will also ensure that any application programs that attempt to use the removed library function will cause a system error.
2. Remove the library function's `#pragma map` and `#pragma linkage` cards from the `tpflink.h` header file.

   Although this simple update is all that is required to block references to the library function at compile time, you can also want to remove the allocator cards from your allocator input deck, the SPPBLD card from SPPGML, and the LOADER CALL card from the appropriate load decks.
3. Ensure that all application programs that called the library function have been altered and recompiled.

   Caution: If any program has coded an explicit `#pragma linkage` it will still be able to refer to the library function at compile time even though it has been removed from the `tpflink.h` header file. Also, if the name of the library function is exactly 4 characters, the compiler will not flag the references to the library function as errors; it will assume the default linkage (TPF Enter/Back) should be used instead of the quick enter linkage. In these special cases, be careful to scan through the source code of your application programs.
4. Load the new version of C000 along with any altered application programs, to your TPF system.

Index

File Number: S370/30XX-40
Program Number: 5748-T14

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SH31-0132-13