MANUAL

Plink II

Lifeboat Associates

THE SOFTWARE SUPER-MARKET

1651 Third Ave., New York, N.Y. 10028 (212) 860-0300 Telex 640693

PHOENIX SOFTWARE ASSOCIATES LTD.

Plink-II

User's Manual

January 15, 1981

Written by Dave Hirschman

Lifeboat Associates
1651 Third Avenue
New York, New York 10028
Telephone (212) 860-0300 Telex 640693

Table of Contents

1 Introduction

Plink-II is a Phoenix Software Associates Ltd. software system that can bind together individually compiled modules of a program into a single file that may be loaded and executed by the operating system. It runs on a Z80 (*) micro-processor under any of the CP/M (*) compatible operating systems.

Plink-II will accept PSA and MicroSoft relocatable files as input (some MicroSoft variants are accepted also: see Appendix). Plink-II offers complete control over the memory utilization of the program, including the ability to create arbitrary overlay structures.

Library files consisting of many separately compiled modules may be used as input. Plink-II offers methods for including all or only the required modules of a library into the program.

Plink-II is a two-pass linkage editor. That is, each of the input files is read twice. Since the output file is built up on disk, Plink-II has the ability to create 64K programs that completely fill the address space of the machine, unlike other linkage editors which require that the output file share memory space with them and their tables. It is even possible to use a small machine to create programs intended for execution on a larger one. A double disk buffering scheme insures adequate execution speed. When overlays are used, programs consisting of up to 4 megabytes of code may be created.

There are many advantages to the practice of linking together separately compiled modules instead of working with a single, large program. A large program may be decomposed into small modules which may be edited and compiled more quickly. Indeed, most compilers available for the Z80 can only handle modules of a limited size, and create output files that require linking before use.

For example, to correct a bug, the programmer need only re-compile the affected modules and re-link the program, instead of re-compiling the entire program. The linkage process is much faster than compilation.

It often happens that a routine is used in several programs, a special I/O routine or COSINE function, for example. Instead of copying the source code for this routine into each program, it may be compiled once and then linked in wherever it may be required. Furthermore, using Plink-II, routines generated with different compilers may be combined into a single program. The programmer can gradually build up a library of general purpose routines and avoid the useless effort of solving the same programming problems over and over again.

Using a linkage editor eases the problems encountered when several programmers must work together on a large program. Once the programming problem has been broken down into separate modules, each programmer may work relatively independently.

The remainder of this guide describes how to use Plink-II. An overview of Plink-II concepts and operation is offered in section 2. The following sections describe Plink-II input statements. The manual is designed to be read sequentially, with side-issues deferred to appendices. The reader should have as background some simple programming experience.

(*) Z80 is a trademark of Zilog.
(*) CP/M is a trademark of Digital Research Corporation.

2 Overview of Plink-II Concepts

Plink-II accepts as input RELOCATABLE or REL files. These are the files that are created by most compilers and assemblers. The term "relocatable" refers to the fact that the machine code in these files may be "relocated" by the linkage editor; that is, the code can be modified to execute at any memory address in the computer. Thus, the programmer is freed from having to choose memory locations himself, and the same code can be loaded at different addresses in different programs.

Each REL file may contain one or more MODULEs. A MODULE is the relocatable code output by the compiler or assembler for a program unit (e.g. Fortran subroutine). REL files containing more than one module are referred to as LIBRARYs. Typically, a manufacturer of a compiler will supply a Library along with it which contains I/O, math, and other support routines needed by the programs compiled in that language. Since the routines are kept in separate modules, Plink can select only those modules that are required by a particular program, thus keeping program size to a minumum.

Each module has a name. Modules with the same name may be included in the same program, but there are several statements that use the module name, and these will refer to a randomly selected module if there are duplicates. In the PSA Assembler, the .IDENT pseudo operation is used to declare the module name. Its use is highly recommended, as the default module name is ".MAIN.", and duplicate module names will result if more than one of these modules is used. The available compilers for the Z80 all generate a module name automatically.

Each module is made up of SEGMENTs (also called "relocation bases"). Segments are the basic units of code and/or data involved in the linkage edit. After Plink-II is aware of what modules are to be included in the program, it assigns memory addresses to each segment in each module. Any code in each segment is relocated so that it will execute at the address to which it is assigned.

Several kinds of segments may be contained in a module, and they all have names that may be referred to during the linkage edit. The main code segment, usually containing all of the executable code in the module, has the same name as the module itself. The main data segment of each module also has the same name as the module, preceded by a double quote ("). For example, a module named ARCTAN would contain a code segment named ARCTAN and a data segment named "ARCTAN (here duplicate module names result in duplicate segment names).

All of the other segments in each module are COMMON blocks. One of these segments is named ".BLNK.", and is referred to as the "unlabeled common". This is the common block that will be created by FORTRAN compilers when the programmer doesn't supply a

specific name for a common block. Other common blocks have names specified by the programmer.

Common blocks typically contain only data, and by default, all common blocks of the same name share the same memory space. In other words, this is a method for defining a block of data that may be shared among several modules. Unlike many other linkage editors, Plink-II doesn't require commons of the same name to be of the same size, or require that the module declaring the largest size be linked first: it simply allocates the maximum space needed.

The CONCATENATE statement changes a common block so that the memory space is not shared: each module is allocated its own space within the common. This is useful for tasks such as creating tables where each module contributes a single table entry.

A SYMBOL is a name given to a particular address within a segment (a RELATIVE symbol) or to some 16-bit number (an ABSOLUTE symbol). One of the major features of the linkage edit process is that each separately compiled module may access symbols defined in other modules. An INTERNAL symbol is one whose address is available to modules other than the one in which it is defined. Module symbols which are not INTERNAL are invisible to other modules. An EXTERNAL symbol is one which is used in a module, but is defined elsewhere. All EXTERNAL references must be satisfied by INTERNAL declarations in another module, or the symbols must be defined in some other fashion.

An ENTRY point is an INTERNAL symbol which comes into play during a library search. In this mode of operation only those library modules having ENTRY points which are referenced as EXTERNAL symbols by one or more already linked modules are included in the program (see the SEARCH and LIBRARY statements).

Duplicate symbols (having the same name) are allowed, but the first definition encountered is used. A warning message is given for the others.

Plink-II offers a great deal of control over how the module segments are arranged in memory. By default, segments are allocated to the next available space, but the LOCATE statement may be used to cause subsequent allocations to be made at a higher address. Segments that form a contiguous unit in memory are organized into SECTIONS. Also, each program overlay is a SECTION.

To sum up, the output of a compiler or assembler is a MODULE. One or modules make up a REL file. Plink-II decomposes the MODULEs into SEGMENTS, re-groups the SEGMENTS into SECTIONS, and organizes the SECTIONS into the output program. In the following section, the commands required to carry out these tasks are described in detail.

3 Plink-II Input Format

The following sections describe some basic input elements. Later sections show how these are combined to create full statements. An appendix gives an abbreviated BNF syntax for those familiar with formal languages.

3.1 Literals

A LITERAL is a sequence of characters delimited by single quotes ('). If a quote may be placed in a literal by using two quotes. Here are some valid literals:

```
'This is a literal'
'This literal contains a '' character'
```

Literals are limited to a length of 128 characters, and must be entered on a single input line.

3.2 16-bit Values

A 16-bit value may be expressed as a number, or as a literal of one or two characters, for example: 'V1'. If a single character is used, its value is formed by using the character as the least significant byte of a word quantity, with zero as the most significant byte. In other words, 'A' would become 0041 hex. When two characters are used, the first becomes the MSB, and the second, the LSB: 'AB' becomes 4142 hex.

A number may be expressed in several different bases, as shown in the table below. A radix character immediately following the number indicates which number system is being used:

Base	Radix	Valid Digits	Valid Range
hex	H	0-9 , A-F	0 - 0FFFF
decimal	.	0-9	0 - 65535
octal	O	0-7	0 - 177777
binary	B	0 and 1	16 digits

If the trailing radix character is omitted, "H" (hex) is assumed. All numbers must begin with a numeric digit (0-9). A preceding minus sign indicates a negative number. In this case, a two's complement representation is used.

The following are examples of 16-bit values:

```
14170     - an octal number
0C1B5     - a hex number
-55.      - a negative decimal number
'A'       - a one character literal
11B       - 11 binary
11BH      - 11B hex
```

The following are not valid 16-bit values:

```
100000.   - decimal number too large
960       - invalid octal digit
'AB       - missing closing quote
C1C2      - does not begin with a digit
```

3.3 Local Variables

Plink-II offers 50 variables that can store 16-bit values during the linkage edit, and may be used wherever a 16-bit value is needed. These are referenced as "#n", where n is a value in the range 1 - 50. For instance, "#1" is a reference to the first local variable.

Note that any 16-bit value will do for n. In particular, another local variable may be used. For instance, if #1 equals 2 and #2 equals 10, the value of ##1 is 10.

To assign a value to a local variable, enter #n := <value>, as shown below:

```
#1 := 10
#2 := 5 + 5
```

Local Variables become useful when the command file requests input from the operator during the linkage edit (see section 8).

3.4 Expressions

Plink-II supports simple, unparenthesized expressions of the form: <16-bit value> <op> <16-bit value>. The following operations are offered:

```
+ :  addition
- :  subtraction
* :  multiplication
/ :  integer division
\ :  remainder after division
& :  16 bit logical and
! :  16 bit logical or
```

The minus sign may also be used as a unary minus (negative). Here are some examples:

```
1 + 1
-0C15H
#5 * 10           (contents of local 5 times 10H)
#1 & 00011000B    (mask contents of local 1)
```

3.5 Identifiers

An identifier is the name of some object, such as a module or segment. A simple identifier is a sequence of no more than eight characters containing no spaces, and containing none of the following:

^=;<>/,\!'#&*+-:@ DEL

Note that an identifier for the data segment of a module (as discussed in the previous section) is preceded by a double quote ("), and can therefore reach nine characters in length. Lower case letters, when used, are automatically translated into upper case. The first character of an identifier may not be a digit 0 - 9.

A concatenated identifier may be created by following the character string by a "^" character, and then entering a local variable designation. The value in the local variable is converted to an ascii string in decimal (base 10) notation, with leading zeros truncated, and is appended to the end of the identifier string. For instance, if #3 contains the value ten, "LABEL^#3" would be treated the same as "LABEL10". The total length of the generated label must be eight characters or less.

The following are examples of valid identifiers:

```
ProgramI
SORT3
"SORT3     (a data segment name)
FOO$$$
BRANCH^#1 (valid only if #1 =< 99)
```

The following are not valid identifiers:

```
34ABC      - begins with a number
CHECKERSI  - too many characters
NIM A      - contains a space
PROG-1     - contains an illegal character
```

3.6 Disk File Names

Plink II adapts itself to the file name format used by the operating system it is executing under. However, it assumes that the characters

^?=_;<>/, DEL

do not appear in file names. In this manual, CP/M format file names are used for purposes of discussion. These file names are of the form [device:]name[.type], with optional portions in brackets. Here are some examples:

PROG1.COM

```
B:CHESS.PRG
SCANNER
```

When the "device:" is not used, Plink-II assumes that the currently logged-in disk is to be used. However, in the case of input files, Plink-II will look for a file on drive A if the file is not found on the logged-in drive. When the PDOS operating system is used, Plink-II will also look in user #5, drive A, in an effort to find the file.

3.7 Initiating Plink-II

Plink-II may be used interactively, or input may be given as it is executed:

Plink-II <statements> <cr>

where <cr> means to press the RETURN key. This means that Plink-II may used in submit files.

To use Plink-II in the interactive mode, enter

Plink-II <cr>

on the console. Plink-II will read commands from the console, prompting with an asterisk "*". All input is stored uninspected until a carriage return is typed. The standard line editing features (rubout, backspace, CTRL-U, CTRL-C, CTRL-E, etc.) supplied by the operating system are available.

A disk file containing all or only part of a command may be inserted into the input at any point by preceding the disk file name with an "@". The default file type is ".LNK". These disk files may not contain further "@" specifications. The most common use of this feature is to prepare a file containing a complete command; then,

Plink-II @<file name> <cr>

links the program. Usually, these ".LNK" files may be prepared once for a given program and used over and over again, greatly simplifying the whole process.

Plink-II reads an entire command, checking for syntax only, before any file processing is done. Then, the program is created before the next command is read.

3.8 Command Format

All Plink-II commands have the same format, regardless of whether the interactive mode is used. Commands are separated by a semi-colon ";". Plink-II terminates when it receives the "Q" command (quit). For example,

<command> ; <command> ; <command> ; Q

Plink-II also terminates when input provided with its execution is exhausted.

All input is free format. Blank lines are ignored, and a command may extend to any number of lines. Comments may be included with input from any source by using a percent sign "%". When this is encountered, all remaining characters on the same line are ignored.

If a CTRL-C is typed while Plink-II is running, it will quit and return to the operating system. If CTRL-E is typed, the current command is aborted, and Plink-II will prompt for more input if it is being used interactively.

Each command to Plink-II links one program, and is a list of statements:

<statement> <statement> ... <statement>

The statements that make up a command typically begin with a key word, and many are followed by arguments separated by commas. For example, in

FILE A,B,C

FILE is the key word, and A, B, and C are the arguments. Key words may be abbreviated by omitting trailing characters, as long as the abbreviation is unique among the entire group of key words. For instance, the previous statement could have been entered as

FI A,B,C

but it could not be entered as

F A,B,C

since this abbreviation could be confused with the FREEMEMORY command.

If the program is linked successfully, its name is printed on the console, along with the address of the highest byte in memory used in the program and the program memory size rounded up to the nearest K (1K = 1024 bytes).

If a syntax error is found, the current input line is echoed with two question marks inserted after the point at which the error was detected. This is followed by an error message (see Appendix). The command must then be re-entered.

If some other error occurs, the linkage edit terminates with an error message also listed in the appendix.

4 General Purpose Statements

4.1 OUTPUT

The OUTPUT statement gives the name of the file that will be created to hold the linked program. The file type, if given, indicates what kind of file is to be produced. It must be one of the following:

PRG - PSA code file, possibly containing overlays. These may be executed directly under PSA's operating system PDOS. A utility program (EXECUTE) is provided to execute them under other operating systems (see appendix).

COM - Absolute binary core-image file, ready to be loaded and executed by the operating system.

If the file type is not given, it defaults to PRG if Plink-II is running under the PSA operating system and to COM for others. If the OUTPUT statement is not used at all, the name of the first input file specified is used, with a default file type. An error will occur if the selected file type does not agree with other statements in the command (for example, overlays in a .COM output file).

The output file replaces any existing file of the same name.

Examples:

```
OUTPUT   PROG1
OUTPUT   PROG2.COM
```

4.2 PROGID

This statement may be used to set the program ID, version number, and revision number of a .PRG output file header (see appendix). The format is:

```
PROGID <name> [,<version> [,<revision>]]
```

with optional input indicated by brackets []. The <name> is a 6 character identifier, and the <version> and <revision> are values lying in the range 0 - 255. The <name> defaults to blanks, and the version and revision numbers default to zero.

Examples:

```
PROGID CHESS,2,1
PROGID TEST1
PROGID PROG5,10
```

4.3 PDOS

This statement has an effect only when a .PRG file is being created. It sets the minimum version and revision of the operating system (PDOS) that must be used to execute the program. Its syntax is:

PDOS <version> [,<revision>]

The version and revision values must lie in the range 0-255, and the revision number defaults to zero if not entered. If the PDOS statement is not used, zero is assumed for both values, meaning that the program will run under any PDOS operating system. The purpose of the statement is to prevent programs from being executed which rely on new operating system features.

Examples:

```
PDOS 2
PDOS 1,9
```

4.4 MAP

The MAP statement may be used to obtain various reports which describe the output of the linkage edit. Reports can be selected that show the memory addresses assigned by Plink-II to the sections, segments and symbols in the linked program, or that describe the modules that were included.

The format of the MAP statement is:

MAP <flag 1> ,<flag 2> ... ,<flag n>

The <flag>s select the desired reports, as follows:

G - Global symbols (i.e. all internal symbols of all loaded modules). The symbols are listed in alphabetical order, with their assigned addresses.

S - Segments. All of the program segments are listed in alphabetical order, and the assigned address and size is given for each.

A - All. This option lists the entire output program, organized into sections. The sections are listed in address order, except where overlays make this impossible. The segments of each section, and the symbols in each segment, are listed in order of ascending memory address. Symbols created via the DEFINE statement or automatically supplied by Plink-II are listed separately.

M - Modules. Each module is listed, along with its ID number, version and revision number, and date and time assembled. This information is available only for PSA format modules. It is supplied by the .PROGID pseudo op in the PSA assembler.

If no <flag>s are given, "MAP A" is assumed. Examples:

```
MAP M,S,G
MAP
MAP S
```

4.5 PAGE

This statement may be used to set the page width of the memory map reports. For example,

```
PAGE 132
```

sets the page width to 132 characters. The various report generators make use of available page width by changing the number of columns per line.

4.6 DATE

Plink-II uses the current calendar date in map report headings and to tag .PRG output file headers with the creation date. If the operating system is able to supply the date, and it looks reasonable, Plink-II will use it. If it is not reasonable, the operator is asked to enter it (it must be entered character for character as shown in the prompt message).

If the operating system can't supply the date, blanks are used unless the DATE statement appears in the input. The operator will then be prompted to enter it as described above. The DATE statement has no arguments:

```
DATE
```

4.7 REPORT

Normally, all memory map reports are written to the list device. This statement can be used to have them written to a disk file instead. A file name may be specified if desired, PRECEDED BY AN EQUAL SIGN. Otherwise, a file is created having the same name as the output program but with a type of .MAP.

Examples:

```
REPORT
REPORT=BUDGET.LST
REPORT = EXPENSE
```

4.8 MAIN

Plink-II requires that a starting address for the program be defined. The starting address is part of the file header in a .PRG file (see appendix), and PDOS transfers control to that address after loading the program into memory. Execution of a .COM file is assumed to begin at address 100H, so Plink-II places a jump at this location to the true starting address of the program.

The starting address is always represented by a symbol named ".MAIN.". If this symbol appears in the program modules, that one will be used. Otherwise, Plink-II will create it, and obtain a starting address from the first input module that has one. The MAIN statement may be used to override this default action and specify which module is the main module of the program. Plink-II will then look only at this module in order to find a starting address.

Example:

```
MAIN PROG1
```

If the MAIN module does not have a starting address Plink-II assumes that it is at the front of the module.

A starting address is defined in the PSA Assembler by supplying a label with the ".END" pseudo op. Most compilers specify a starting address for the main module automatically, so this statement usually need not be used (for exceptions, see "MicroSoft Hints" appendix). The starting address may be in an overlay if desired.

4.9 DEFINE

This option may be used to give values to symbols which are not defined by any module in the program. These defined symbols are then used to resolve EXTERNAL references made by the program modules. The symbols can be given absolute values, or may be defined as a plus or minus offset to some other symbol. For example:

```
DEFINE CONST1=1238., FLAGS = 10110011B, COUNT = #5,
                S1 = 10, S2 = S1, S3 = S2 + 5
```

Note that when a symbol is defined as an offset to another, the expression following the plus or minus sign is evaluated separately. For instance,

```
DEFINE A = B - 2 + 3
```

is equivalent to

```
DEFINE A = B - 5
```

There are some symbols which are pre-defined by Plink-II. A list of them is given in Appendix A.

4.10 CONCATENATE

As discussed in the overview, module segments other than the program and data segments are treated as common blocks, so that each module shares the same memory space. The CONCATENATE statement changes a common block into a "concatenated segment". In these, each module is allocated its own memory space and no sharing takes place. However, the space used by all modules is concatenated together (in the order encountered) to form a single segment. This statement is useful for tasks such as constructing a table where each module contributes one table entry.

The segments to be concatenated are listed, separated by commas:

```
CONCATENATE COMMON1, COMMON2
```

5 Defining Program Structure

In this section the statements that cause particular files and modules to be linked into the program and define the memory structure of the program are described. As discussed in the overview, Plink-II groups program segments having similar characteristics into SECTIONs. In other words, segments are loaded into the same section until something like a change of memory address or an instruction to start an overlay area is encountered. Then, a new section is begun.

5.1 FILE, LIBRARY, SEARCH

These statements are used to define the .REL files and librarys that will be the input to the linkage edit. Each of them is followed by a list of .REL file names, separated by commas:

```
FILE    MAIN, PASS1.REL, PASS2
SEARCH  PASLIB
LIBRARY MATH.LIB, APPLIB
```

The default file type for these files is .REL. All of the modules contained in files listed in the FILE statement are included in the output program (except for those specifically eliminated by a following statement). When the LIBRARY statement is used, only those modules defining symbols that were used by modules already linked, but not defined yet, are selected. This selection process is known as a "library search", and is commonly used for the runtime support librarys supplied with most compilers: the size of the program is reduced, because only those parts of the runtime support that are actually needed are loaded.

The SEARCH statement is the same as LIBRARY except that Plink-II may make multiple passes through the file if undefined symbols remain even after all specified files have been read. This capability is provided because sometimes a library module has undefined symbols that are defined only by modules that have already been passed over. It should hardly ever be necessary to use it: most librarys are set up to be loaded with a single pass.

The modules encountered in files loaded with the FILE, LIBRARY, and SEARCH statements are normally assigned to the current section, but other statements (described later) can override this default action.

5.2 INCLUDE, EXCLUDE

These statements apply to the file last mentioned in a FILE, LIBRARY, or SEARCH statement, and are used to eliminate one or more modules from the linkage edit. The modules are listed, separated by commas:

```
INCLUDE MOD1,MOD2
EXCLUDE MOD3
```

The EXCLUDE statement prevents the named modules from being included in the linkage edit. When the INCLUDE statement is used, all modules except the ones named are excluded. This action occurs prior to any library search selection. INCLUDE and EXCLUDE may not be used on the same file.

5.3 MODULE, SEGMENT

When the FILE, LIBRARY and SEARCH statements are used, all of the selected modules are usually loaded into the current section. These statement override this default action, and place given modules or segments into the current section. The module or segment names are listed, separated by commas:

```
MODULE  MAIN, MOD5
SEGMENT MAIN, "MAIN, MOD5
```

The SEGMENT statement may be used on any segment. The MODULE statement should be used only on modules, and is the equivalent of a SEGMENT statement on the program and data (if any) segments of the module. For instance, in the above example the two statements have the same effect if MOD5 has no data segment.

5.4 SECTION

The SECTION statement is used to give an 8 character name to the current section:

```
SECTION global
```

The name is used for memory map reports. If a name of .ROOT. is used however, this statement has the effect of specifiying the main section of the program. This is the section that will be loaded from the .PRG file when the program is executed. Any other sections are loaded by the overlay handler (see discussion of overlays following).

5.5 NOSORT

Normally, the segments within each section are sorted so that uninitialized ones are grouped at the end (an uninitialized segment is one that may reserve memory space, but has no code or data in it). Sorting a section this way minimizes the disk space required to store it, since no space is needed for the uninitialized segments at the end. Note that the program must be careful about initializing data in this case, because garbage is left in memory in the uninitialized places when the section is loaded. If this is a problem, or if the user requires for other reasons that the segments within a section be left in input order, the NOSORT statement may be used. It applies to the

current section only, and has no arguments.

Aside from moving the uninitialized segments to the end, the sorter leaves the segments in the order encountered in the input files. However, the Fortran blank common .BLNK. is sorted last (if uninitialized). This is done to facilitate the access of Fortran programs to free memory.

5.6 .DATA.

Occasionally it is convenient to group together all data segments of a program into a single section, for example, to create a ROMable program. The .DATA. statement, which has no arguments, is used for this purpose. When encountered, a separate section is created, and all data segments are assigned to it, unless the MODULE or SEGMENT statements are used on them. These statements override the FILE, LIBRARY and SEARCH statements.

5.7 SCTEND

This statement may be used to end the current section and cause a new one to be started. It has no arguments.

5.8 LOCATE

Normally, Plink-II assigns memory addresses sequentially to segments as they are encountered in the input files. This statement changes the address where subsequent memory allocation will occur, and begins a new section. The new address must be greater than or equal to the current allocation address.

Example:

```
FILE   PROG1,PROG2
LOCATE 4000H
MODULE MOD1
```

In this example, the modules in files PROG1 and PROG2 are loaded at the beginning of available memory, but module MOD1 is loaded at address 4000H, even if it is contained in PROG1 or PROG2.

Another handy use for the LOCATE statement is the creation of a "patch area" within a program:

```
OUT    PROG.COM
LOCATE 200
FILE   PROG1,PROG2,PROG2
```

This program has a "hole" starting at address 107H and extending up to address 200H, which could be used for patches (see appendix for .COM file format).

When a COM file is being output, "holes" created with the LOCATE statement appear in the disk file as well as in memory: a COM disk file is always an exact image of what the program will be when loaded into memory. When a PRG file is created, however, the hole is not present in the disk file. Only the space needed by each section is allocated, and when the program is executed, only the "global" section is loaded into memory by the operating system: any other sections are automatically loaded by initialization code generated by Plink-II, using the overlay handler (see discussion of overlays following). The global section is the usually the first non-overlaid section, or may be specified by using the SECTION statement.

5.9 ACTUAL

As discussed in section 2, each program segment is normally relocated to execute at the memory address at which it is to be loaded. When this statement is used, however, a new section is begun, and all segments within it will be assigned execution addresses starting at the given value. This includes changing the value of all addresses referenced from inside or outside those segments (presumably, the segments will be moved at run time to the correct execution address). The ACTUAL applies only to the new section (for instance, it can be turned off via a SCTEND statement)

For example, suppose it is necessary to create a program which will run on a non-CP/M type system which loads programs at address 4000H:

```
OUT     PROG.COM
ACTUAL  4000H
FILE    PROG.REL
```

Here, a .COM file is created having the code for PROG.REL at the front, but the addresses within the code are adjusted to execute at 4000H. This feature is also ideal for tasks such as the creation of bootstrap routines and I/O drivers which have to be moved to non-standard addresses before use.

5.10 FREEMEMORY

Often, a program requires the address of any free memory left over after the program is loaded (for example, the PASCAL heap). Plink-II will use the first byte following the program for the free memory address by default, but this statement may be used to put the free memory at the current allocation address instead. For example,

```
FILE        MAIN, MOD1, MOD2, MOD3
FREEMEMORY
FILE        INIT
```

Here, the space used by the initialization code in file INIT is re-used as free memory when the main program begins execution. FREEMEMORY begins a new section, which in this case starts with INIT.

Plink-II creates a symbol named ".END." which has as its value the free memory address. Also, the words at symbols $MEMRY and ?MEMRY are initialized to this value (if the symbols exist) for MicroSoft and Digital Research PL/1 programs.

5.11 Creating Overlays

If a program is too large to fit into memory, OVERLAYs must be used. Overlays are simply sections of the program which are set up to use the same memory area. When overlays are specified, an overlay loader module is automatically included into the program. The loader reads the overlays from disk into memory automatically as required by the executing program. Since portions of the program share the same memory space as the program runs, the memory requirment of the program is reduced. The disadvantage, of course, is that the program runs more slowly due to the extra time needed to load the overlays from disk.

These statements are used to define overlay structures. Normally, arbitrarily complex overlay structures may be created with no modifications to the program modules, but some rules concerning calling sequences and accessing data must be obeyed. Also, the overlay structure should be organized to minimize the number of times overlays have to be loaded in: if a program loop that is executed 100,000 times has to switch from one overlay to another each time, and it takes .05 seconds to load an overlay, the program will run for almost three hours! Some of these issues will be discussed more fully later on.

An OVERLAY AREA is a group of overlays which share the same memory address. To create an overlay area beginning at the current memory allocation address, the BEGINAREA statement is used (no arguments). Each overlay area must be ended by an ENDAREA statement. Each overlay within the overlay area is begun by an OVERLAY statement. This is followed by FILE, LIBRARY, SEARCH, MODULE, or SEGMENT statements to place the desired items into the overlay.

Here is a simple example:

```
OUTPUT  TEST.PRG
FILE    F1
BEGIN   OVERLAY FILE F2
        OVERLAY FILE F3
        OVERLAY FILE F4 END
```

This command creates a program named TEST.PRG. When it is loaded by the operating system (or the EXECUTE utility if a non-PDOS operating system is being used) only the code in file F1 is loaded: this is referred to as the RESIDENT section of the program. Then execution begins. Suppose a call is made to some code from file F2. The overlay loader is automatically invoked to read that overlay from the .PRG file, and then a branch is made to the called routine. Later, if a call is made to code in F3, for example, F3 will be loaded, over-writing F2. If F2 is called again before F3 or F4, however, no disk I/O need be done since the required overlay is still in memory. Note that overlays are not stored back on disk when they are over-written, and a fresh copy is loaded the next time the overlay is needed: the PRG file is never modified.

Plink-II accomplishes these tasks by replacing an address pointing to an overlaid routine with a call to a small piece of code which calls the overlay loader, to insure that the proper overlay is in memory, and then jumps to the overlaid routine. This piece of code is called an OVERLAY VECTOR. Note that a program can either call or jump to an overlaid routine. However, the user must insure that when a return to the caller is made, that the caller is still in memory. In the previous example, for instance, if F2 calls F3, F2 will be smashed by F3 before the return to F2 is made, resulting in a program bug. F2 could jump to F3 with no problem, however.

Note that Plink has no way of knowing whether an address in an overlay represents code or data. In this example, for instance, if F2 references F3, it is always assumed to be a call or jump: data can't be accessed from one overlay to another in the same area. A more exact rule for how data may be accessed within overlays will be given shortly.

In order to more easily discuss overlay structures, Memory Diagrams such as figure 1 will be used. In these diagrams, the vertical dimension represents memory addresses, with lower addresses at the bottom. The horizontal dimension is used to indicate where memory locations are shared. In figure 1, for instance, F1 reaches all the way from left to right because it shares its memory with no one: it is resident. F2, F3, and F4 share the same memory space within the overlay area. Since F3 is larger than F2 or F4, some memory is unused when these overlays are in memory. These wasted areas are shaded in.

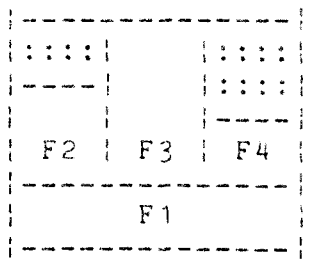


Figure 1

More than one overlay area may be created. Any additional ones are allocated to the next available space in memory, as in the following command:

```
OUTPUT TEST2
FILE F1, F2, F3, F4, F5, F6, F7
BEGIN OVERLAY SEG MOD1
      OVERLAY SEG MOD2A, MOD2B, MOD2C
      OVERLAY SEG MOD3 END
BEGIN OVERLAY MOD MOD4
      OVERLAY SEG MOD5
      OVERLAY SEG MOD6 END
```

Here two independant overlay areas are created. One overlay from the first and one from the second may be in memory simultaneously, as shown in Figure 2. Another feature of this command should be noted. All of the input files are simply listed in the beginning, and the desired segments to be overlaid are pulled out via the SEGment or MODule commands. Therefore, most of the data segments and common blocks will be allocated to the resident section. This is usually desirable, as any data in an overlay may be smashed when another overlay overlapping it in memory is read in. Another way of accomplishing this task is by using the .DATA. statement outside of the overlay areas: this will automatically pull all of the data segments and common blocks out of the overlays. If it is desired to have a data segment assigned to the same overlay as its related program segement, the MODULE statement could be used instead of SEGMENT, as shown for MOD4.

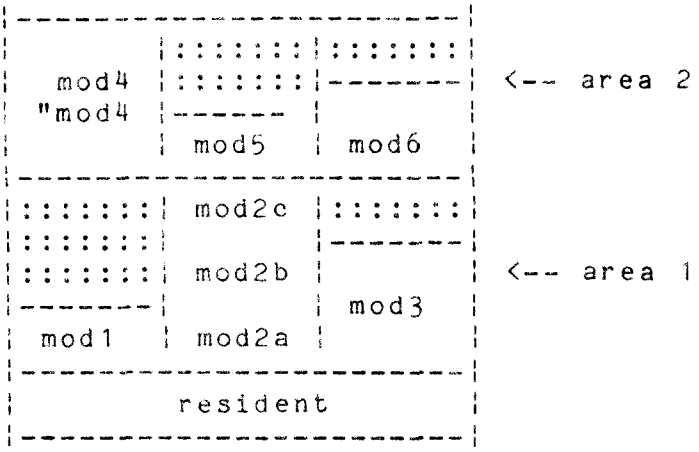


Figure 2

Notice that in Figure 2, the second overlay area starts in memory after the largest overlay in the first area. The SHARE statement can be used to cause Plink-II to ignore the following item when calculating the maximum size of an overlay area. Suppose, in the example of Figure 2, that Mods 4 - 6 will never be accessed when Mods 2a - 2c are in memory. Then the program could be made slightly smaller as follows:

```
OUTPUT TEST2A
FILE F1, F2, F3, F4, F5, F6, F7
BEGIN OVERLAY SEG MOD1
      SHARE OVERLAY SEG MOD2A, MOD2B, MOD2C
      OVERLAY SEG MOD3 END
BEGIN OVERLAY MOD MOD4
      OVERLAY SEG MOD5
      OVERLAY SEG MOD6 END
```

The resulting memory organization would be as shown in Figure 2A. When the Mod2a - c overlay is in memory, none of the others may be. When Mod1 or Mod 3 are in memory, one of Mods 4 - 6 may be in memory simultaneously.

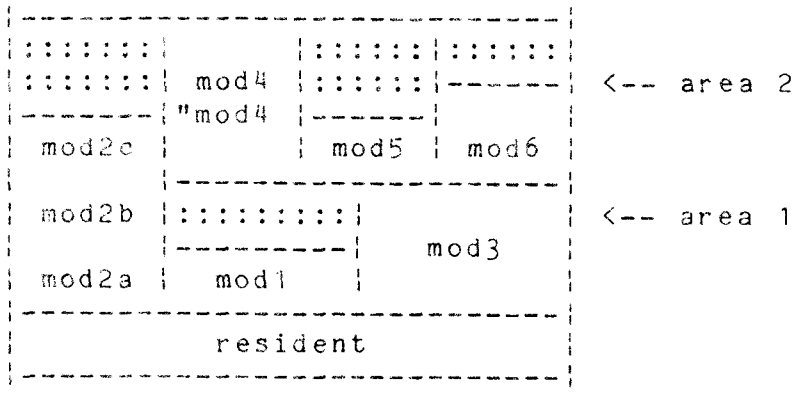


Figure 2a

Another mechanism for creating overlay structures results from the fact that the BEGINAREA and ENDAREA statements may be nested to create overlay structures up to 32 levels deep, as shown in the following example:

```
OUTPUT TEST3
BEGIN OVERLAY SEG MOD1
      OVERLAY SEG MOD2 BEGIN OVERLAY SEG MOD4
                             OVERLAY SEG MOD5
                             OVERLAY SEG MOD6 END
      OVERLAY SEG MOD3 END
FILE F1, F2, F3, F4, F5, F6, F7
```

In order to discuss this overlay structure (see figure 3), some new terminology must be developed. Every section of a program is assigned a LEVEL NUMBER. Resident sections are assigned a level number of zero. Whenever a BEGINAREA command is entered, the level number is increased by one, and it is decreased by one at an ENDAREA statement. In Figure 1, F1 is level zero, while F2, F3 and F4 are level 1. In Figure 3, MODs 1 thru 3 are level one, and MODs 4 thru 5 are at level 2.

Every overlaid section has an ANCESTOR. An overlay's ancestor is the last section defined prior to the beginning of the overlay area and at a lower level number. In figure 1, F1 is the ancestor of F2, F3 and F4. In figure 2, the global section is the ancestor of all the overlays. In figure 3, the global section is the ancestor of MODs 1 thru 2,3 while MOD2 is the ancestor of MODs 4 thru 6

The ANCESTORS of an overlay consist of the overlay's ancestor, the ancestor of that overlay, and so on, until a resident section is reached. Notice that the number of ancestors an overlay has is equal to its level number. For instance, in figure 3, the ancestors of MOD5, which is at level 2, are MOD2 and the resident section.

Conversely, a section's DESCENDANTS consist of all those sections which have it as an ancestor. The descendants of MOD2 are MODs 4 thru 6.

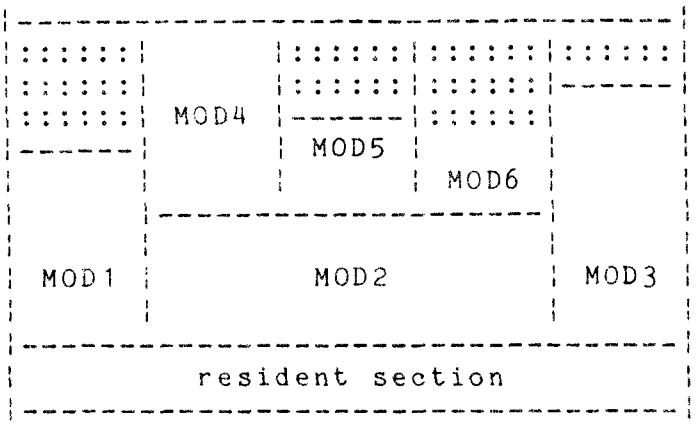


Figure 3

Now the rule for accessing data within overlays can be explicitly stated: data within an overlay may be accessed only from within the overlay, or from one of the overlay's descendants. For instance, data within MOD2 may be accessed only from within MOD2 or from MODs 4 thru 6. The data within MOD1 may be accessed only within MOD1, since it has no descendants.

Plink-II works with the overlay loader to guarantee the "data access rule" by obeying the following rules:

1. Whenever an overlay is in memory, all of its ancestors must be in memory as well. For instance, if in the example of Figure 3 MOD6 were to be called with no call to MOD2, MOD2 would be automatically loaded anyway.

2. The overlay vectors are never used when an overlay accesses a descendant. This means that any instructions which access data in this direction are not modified.

The SHARE statement may be used on an entire overlay area. In this case, the size of the overlay area is not included in the maximum size of the surrounding overlay area.

5.12 Selecting an Overlay Loader

Normally, the overlay loader must reside on disk in a file named OVERLAY.REL. It should not appear in a FILE, LIBRARY or SEARCH statement.

Three versions of the overlay loader are supplied on the Plink-II distribution disk. OVERLAY.REL is the standard loader, and requires a Z80 (*) processor. OVERLAY8.REL is a loader which will run on an 8080 (*) processor. It may be selected by using the I8080 statement, which has no arguments. There is also an

OVERLAYD.REL file. It contains a debugging overlay loader which displays messages on the console indicating what overlaid addresses are being called, and which overlays are being loaded. The memory map A report will be helpful in interpreting these messages. This loader will also execute on an 8080 processor, and may be selected by using the DEBUG statement, with no arguments.

For example, suppose you have a Plink-II command file PROG1.LNK which links some program with overlays, and you wish to create a version of the program that uses the debugging overlay loader. Typing

```
PLINK-II DEBUG @PROG1
```

would create the desired program.

(*) Z80 is a trademark of Zilog.
(*) 8080 is a trademark of Intel Corp.

6 Transfer of Control

Any Plink-II statement may be optionally preceded by a LABEL, which is simply an identifier followed by a colon:

```
L1:  FILE PROG1
```

These labels may be used as branch points with some simple statements described here.

The GOTO statement causes an unconditional branch to the given label:

```
GOTO L1
```

When this statement is used, Plink-II searches forward through the input stream (wherever it is coming from, see section 3) to find the label. If the end of the input is reached without finding it, the input stream is reset to the front, and the search begins again. If input is coming from the console, it is reset to the beginning of the current line only: transfer of control statements are generally useful only when input is coming from a disk file.

Since this search process is rather slow, the BACKTO statement is supplied. It is the same as GOTO, but resets the input stream to the front immediately before beginning the search. It is therefore faster when the label precedes the goto statement.

Either GOTO or BACKTO may be preceded by an IF statement to create a conditional branch. The IF is followed by a simple relational test, as in the following example:

```
IF #1 = 'Y' BACKTO L2
```

The relational test consists of two 16-bit values separated by one or two relational operators from the set =, <, and >, which mean "equals", "less than" and "greater than", respectively. For example, to test for "not equal" one would use "<>".

The transfer of control statements become useful when the input stream performs alternative actions based upon operator input, as described in the next section.

7 Console Input/Output

This section describes statements that can display messages and get input from the console. When these statements are used in an "@" file, in conjunction with the transfer of control statements given in the previous section, Plink-II can be used as a program configuration system that generates different versions of the program based on operator input.

The DISPLAY statement takes as arguments items to be displayed on the console. The following items may be displayed:

literal	- the literal is displayed as is, without the surrounding quotes.
TAB	- a tab character is output.
CR	- a carriage return/linefeed sequence is output.
BELL	- a 07H is output.
16-bit value	- The value is displayed in the current radix.
HEX	- this does not cause any display to take place, but changes the radix that all following 16-bit values are printed in to hexidecimal. This is the default at the start of the DISPLAY statement.
DECIMAL	- this displays nothing, but changes the 16-bit value display radix to decimal. Leading zeroes are truncated.

Here is an example of DISPLAY usage:

```
DISPLAY BELL, 'Linkage edit completed', CR
```

There are three commands for getting values from the operator. They all require a local variable as the first argument, which is where the value will be stored, and optionally take a CR as the second argument. When used, this will cause a carriage-return line-feed to be echoed following the operator's input.

The GETVAL statement inputs a 16-bit value in any of the allowed forms. PROMPT inputs a single character, converts it to upper case, and interprets that as a 16-bit value. GETCHR is the same, but requires the operator to hit the return key after the single character has been entered. This gives the operator a chance to inspect the character and/or correct it.

Here is an example of how the input/output statements might be used. Suppose we have a program that uses some special purpose data communications hardware that comes in two variations. Three modules are written which have the same ENTRY symbols but different code to support the two versions of the hardware, and a third module for testing purposes. A single command can link all three versions of the program, asking the operator which version of the data communications driver is wanted:

```
ASK:
  DISPLAY BELL,'Enter 0 for ACME 510 Modem',
               ' 1 for ABC Electronics Modem',CR
               '2 for self-test loop =>'
  GETVAL #1, CR
  IF #1=0 GOTO ACME510
  IF #1=1 GOTO ABC
  IF #1=2 GOTO SELF
  BACKTO ASK
ACME510: FILE ACME510
  GOTO END
ABC:     FILE ABC
  GOTO END
SELF:    FILE SELFLOOP
END:
```

The correct file is linked depending on the operator's selection. This particular problem could have been handled by having three separate link command files, but what if a system has 10 versions of one module, and 10 of a second module? Then 100 versions of the program are possible, and the interactive methods offered here are clearly advantageous.

Another way to handle the above example is to use the "concatenated ID" mechanism described earlier to create a label to GOTO by appending the operator's response to a label prefix. Also, the three modules to be selected from are combined here into a library, and the INCLUDE statement is used to select the correct one.

```
  FILE DRIVERS.LIB
  GETVAL #1,CR
  IF #1>2 BACKTO ASK
  GOTO LAB^#1
  LAB0: INC ACME10   GOTO END
  LAB1: INC ABC      GOTO END
  LAB2: INC SELFLOOP
END:
```

Appendix A - Pre-Defined Symbols

There are a few global symbols which are pre-defined by Plink-II before the linkage edit begins. They are listed below. The user should not attempt to define these symbols, as naming conflicts will result (except in the case of .MAIN.). Future versions of Plink-II may have more of these symbols. They will be of the form .XXXX., so the use of symbols of this form should be avoided.

.END. - This symbol has as its value the address of free memory. Normally, this will be the address of the first byte above the program, but this default action may be changed via the FREEMEMORY statement.

.MAIN. - This symbol may be defined by the user, or will be defined by Plink-II otherwise. It has as its value the starting address of the program (see the MAIN statement).

.OVLY. - This is the entry point to the overlay loader used by the overlaid symbol vectors.

.LOAD. - This is another overlay loader entry point which is used to load sections of code in .PRG files which are resident but not loaded by the operating system at execution time. It may also be used by the program to cause any overlay to be loaded. The A register must contain the overlay number. These numbers are given in the memory map A report.

.OVEX. - This symbol points to the last instruction in the overlay loader before control is transferred to the overlay. It is useful for debugging (breakpoints can't be set in an overlay until it is present in memory).

.INIT. - This symbol points to the initialization code for a program containing overlays. In such programs, control is passed to the overlay loader's initialization routine after the program is loaded. When the overlay loader is finished initializing itself, it jumps to this label. The linkage editor generates code here to load resident sections of the program not loaded by the operating system (using the .LOAD. entry point in the overlay loader) and then jumps to .MAIN., the true starting address of the program.

In addition to the above pre-defined symbols, if a Microsoft format REL file was included as part of the input, and has defined a global symbol named "$MEMRY", then the value of .END. is stored at that address. The ?MEMRY symbol is handled the same way for Digital Research PL/I programs.

Appendix B - COM and PRG File Format

COM File Format

COM files are the standard executable file accepted by CP/M type operating systems. They are designed to be loaded into memory at address 100H. Therefore, disk addresses equal memory addresses minus 100H. Plink-II places a 7 byte initialization routine at address 100H which loads a pointer to free memory into the stack register and then jumps to the start of the program:

```
LHLD 6
SPHL
JMP  .MAIN.
```

If the initialization routine is not wanted, a "LOCATE 100" statement may be used to start the program at 100H, overwriting it. Hopefully, the first executable instructions of the program would be at the very front.

.PRG File Format

PRG files are output files that can be executed directly only on Phoenix Software Associate's PDOS operating system. A utility program is supplied to execute these programs under CP/M type operating systems (see appendix). A PRG file must be created if overlays are used.

PRG files have a 128 byte header in front which has the following format:

PRG file header

Address (hex)	Size (bytes)	Contents
0	6	Program ID
6	3	Date linked (packed BCD)
9	3	Time " "
C	2	Minimum PDOS version and revision required for execution
E	2	Program version and revision
10	2	Required program memory space
12	2	Memory load address
14	2	Program starting address
16	2	Program load size
18	2	Requested stack address
1A	2	Address of overlay table
1C	100	Reserved for future expansion

When the program is to be executed, the minimum PDOS version and revision number are checked against their current values. Loading continues only if the current version is greater than or equal to the version number given in the header. In this way, programs using new PDOS features are protected against execution under operating systems not having these features.

Next, the total amount of memory needed to run the program is checked against the amount actually available. The execution is aborted if not enough memory is available. This check prevents programs from being executed which might fit into memory initially, but will smash the operating system when overlays are done.

If the program passes these tests, it is loaded from disk (beginning after the header) into memory (at the specified memory load address). Only the amount of the program given in the load size field is loaded: the rest of the .PRG file, if any, consists of overlays.

When loading is complete, the stack register SP is set to the value given in the PRG header. If this value is minus 1, however (FFFFH), SP is loaded from address 6, which contains the highest address available to the program (recall that the Z80 stack grows down in memory). Finally, control is transferred to the given starting address.

Most of the other information in the PRG header may be printed by using the PDOS VERSION command.

Appendix C - More Plink-II Examples

Here, some examples that illustrate a few points not brought up fully in the body of this guide are presented.

Example 1

Suppose you have a program consisting of just one module, contained in file TEST.REL. To produce a file TEST.COM to execute, just type:

```
Plink-II FI TEST<cr>
```

Recall that the name of the output file defaults to the name of the first input file (the only input file in this case). This is a simple link, with no memory map or other options. The module must have a defined starting address, and no external symbols.

Example 2

Suppose a program has been created consisting of three modules: MOD1, MOD2, and MOD3. Each of these modules exists in separate disk files, called MOD1.REL, MOD2.REL and MOD3.REL. MOD1 is the module where execution is to begin, and it has a defined starting address. To create a COM file ready for execution, execute Plink-II, and in response to the prompt, enter the following command:

```
OUT PROG.COM
FI  MOD1, MOD2, MOD3
MAP M A
REPORT;
```

Two memory map reports will be written to disk file PROG.MAP. When Plink-II has finished, enter Q (followed by a carriage return) to terminate it.

Example 3

Suppose that it is desired to add an I/O driver for a line printer to the system. The I/O driver is to be loaded high up in memory, so that it will not interfere with normal user programs. When executed as a normal program, the driver is to automatically load itself into the correct address and stay there until the computer is powered off. This is easily accomplished by using some of Plink-II's special options.

The program will consist of two modules: the driver itself, and a loader module. Suppose that the printer is interfaced through a single port, number 90. An input from this port gives the printer status: a zero indicates that the printer is ready to accept another character, while anything else indicates that the printer is not ready (power off, out of paper, etc.). Characters are printed by writing them to port 90. The driver is used by calling it with a character to be printed in the A register. The following Z80 assembler code makes up the driver module:

```
        .IDENT  DRIVER
        .ENTRY  PRINT
        ;
PRINT:  MOV     C,A             ;SAVE CHARACTER
..WT:   IN      90H             ;WAIT FOR READY
        CPI     0
        JRNZ    ..WT
        ;
        MOV     A,C             ;RESTORE CHARACTER
        OUT     90H             ;OUTPUT IT
        RET
        .END
```

A module is produced with a single entry point, PRINT. Now for the Plink-II input. The following Plink-II command is placed into a file called PRINTER.LNK:

```
OUT  PRINTER.PRG
FILE LOADER
LOC  0F800
FILE DRIVER
MAP
```

Finally, after assembling LOADER and DRIVER, typing:

```
Plink-II @PRINTER
```

causes the desired program, PRINTER.PRG, to be created.

What happens when PRINTER is executed? Since there is a gap between the LOADER and DRIVER modules because of the LOCATE statement, Plink-II includes the overlay handler into the program. Only the LOADER module is loaded by the operating system: initialization code generated by Plink-II calls the overlay loader to load DRIVER. In other words, the LOADER module doesn't have to do anything but return to the operating system! All the rest is taken care of automatically.

If at a later time it is desired to load the printer driver at a different address, this may be accomplished by simply changing the argument to the LOCATE statement. Neither of the modules would have to be re-assembled.

Appendix D - MicroSoft Hints

As described earlier, Plink-II will accept MicroSoft format REL (relocatable) files as input to a linkage edit. The MicroSoft format is rapidly becoming an industry standard for Z80 compiler output. However, several manufacturers are selling compilers which output files that look like MicroSoft's, but actually contain subtle differences. Plink can handle some of these as is, and some can be handled if certain restrictions are maintained, but others will not work. Also, MicroSoft periodically makes minor changes to the format in order to support new language features.

The list below indicates which compilers have been checked out with Plink-II. Usually, lower numbered versions are also handled correctly. If a compiler you wish to use is not on this list, either insure that it outputs a format compatible with a listed one, or contact Phoenix Software Associates.

MicroSoft:
- Cobol 4.01: in 4.01, the SECTION statement may not be used.
- Fortran 3.31: the blank common is automatically renamed to .BLNK. by Plink-II.

Cromemco:
- Cobol 3.01

Digital Research
- PL/1 1.2: the indexed .IRL files are not supported: they will have to be converted to normal form with the LIB program.

MT MicroSystems:
- Pascal MT+ 5.1: .ERL files designed to be input to the disassembler are not supported. Also, the MAIN statement must be used to indicate the main module of the program since these .REL files do not specify the starting address.

When using MicroSoft format files, the following points should be remembered:

1. When overlay structures are created, a common block may not share the same memory space with another common when both are accessed by the same module, or error #93 will result. This is due to the fact that fix up addresses are given in terms of memory addresses, addresses in overlapping commons are ambiguous.

2. A runtime suppory library is provided with most compilers, and it usually must be specified via the LIBRARY command (i.e. it is searched only once). However, some compilers use the automatic library search feature that is offered in the MicroSoft format. In these cases the LIBRARY statement need not be used, although it will do no harm if entered.

Appendix E - PSA Assembler Hints

This appendix is a list of hints which may be of help in setting up PSA Assembler modules for use with Plink-II.

SYMBOLS

Internal and External symbols are created by using the .INTERN and .EXTERN pseudo operations. .ENTRY is used to create entry-point symbols.

SWITCHES

When assembling a module for use with Plink-II, do not use the .PABS or .XLINK switches. Do use the .PREL and .LINK switches (these are defaults). You may use the .PHEX switch to get an ASCII .REL file, but using .PBIN (the default) will result in a savings of disk space.

MODULE NAME

Always use the .IDENT operation to give each module a unique name. If you don't, the modules will all have the name .MAIN., creating naming conflicts if Plink-II statements reference those modules or their program and data segments.

.LOC Pseudo Op

Do not attempt to load some code or data at an absolute address by using the .LOC pseudo op with a number. If you want to load something at an absolute memory address, make it a separate module or common block, and use the LOCATE statement of Plink-II to put it where you want it. Error #50 will occur if an attempt is made to specify an absolute load address in a module.

STARTING ADDRESS

A label should be supplied with the .END pseudo op to define the starting address of the main module of the program. If any other modules to be linked also have a starting address, it is a good idea to use the MAIN statement of Plink-II to indicate the main module.

Alternatively, make the starting address .MAIN., and declare .MAIN. as an .INTERN symbol.

The starting address MAY NOT be an absolute symbol: it has to be relative to some module.

LIBRARIES

Libraries may be created by using the .PRGEND switch. This results in the creation of a new module starting at that point. Alternatively, individual REL files may be concatenated into a single file by most file copy utility programs. One has to be careful, however, that CNTL-Z is used as an end of file condition for ASCII files, but physical end of file is used for BINARY files. Binary and ASCII modules may be mixed together in a library. Microsoft and PSA REL files may not be mixed in the same library.

MEMORY MAP

If the M report of the memory map is wanted, use the .PROGID pseudo op to define the program name, version number, and revision number.

COMMON BLOCKS

To make a common block, declare the common block name to be an .EXTERN in each module that must reference it. The common should not be declared .INTERN by any module. Then, use .LOC to define the common. For example,

```
            .EXTERN TABLE
                    .
                    .
                    .
            .LOC    TABLE
A:          .WORD   5
B:          .BLKB   10
C:          .ASCII  "ABCDEFG"
            .RELOC
```

declares a common named TABLE consisting of A, a word, B, 10 bytes long, and C, an ASCII string. A useful way to handle commons referenced in several different modules is to keep the common definition in a separate file and use the .INSERT statement to include it with each module.

Remember that most FORTRAN compilers will name a common .BLNK. if the programmer does not give it a name.

DATA AREA

Objects are placed into the data segment of a module by preceding them with a .LOC .DATA. The programmer may .LOC .DATA. over and over again in the program: each definition is added on to the end of the previous ones. For example:

```
            .LOC    .DATA.
FOO:        .WORD   0
BAR:        .BYTE   55H
            .RELOC
                    .
                    .
                    .
            .LOC    .DATA.
PTR:        .WORD   TABLE
TABLE:      .BLKW   100
            .RELOC
```

reserves space for four variables in the data segment. Plink-II can be instructed to load data segments and common blocks anywhere in memory via the SEGMENT statement.

Appendix F - The EXECUTE Utility

PRG output files may be executed directly only under Phoenix Software Associates' PDOS operating system. The EXECUTE utility program is provided to allow them to be executed on any CP/M type operating system.

To perform this action, enter EXECUTE followed by the standard command that would be given to execute the program normally. For instance, suppose a program named COPY.PRG is to be executed with two file names as arguments. On a PDOS system, the operator would enter

```
COPY FILEA FILEB
```

but on other systems, the operator would enter

```
EXECUTE COPY FILEA FILEB
```

The EXECUTE program loads the program for execution exactly as PDOS would, including the initialization of all memory areas from address zero to 100 hex.

Sometimes it may be desirable to create a COM file that executes a particular PRG file, to save typing if a program is frequently used, or to create a program for unsophisticated users. EXECUTE performs this function also, by giving as arguments a slash (/) followed by the name of the PRG file that will be executed, and the name of the COM file that will execute that particular PRG file. The COM file name and drive id default to the PRG file counterparts. Continuing with the above example,

```
EXECUTE /COPY COPY
```

would create a specialized version of the EXECUTE utilty named COPY.COM, which would be configured to execute the COPY.PRG file directly. This could have been abbreviated as

```
EXECUTE /COPY
```

since the COM file name defaults to the PRG file name. Note that the drive id given with the COM file name specifies what drive the COM file will be written to (logged-in disk if omitted), while the PRG drive id specifies on what drive the COM file will look for the PRG file (again, logged-in disk if omitted). For example,

```
EXECUTE COPY B:COPY
```

creates a file COPY.COM on the B drive which, when executed, attempts to load COPY.PRG from the logged in disk. On the other hand,

```
EXECUTE A:COPY B:
```

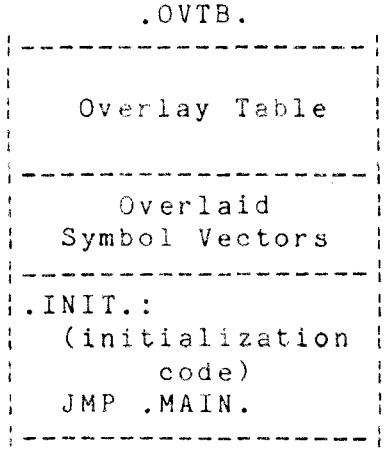
creates a file COPY.COM on the B drive which, when executed, attempts to load COPY.PRG from the A drive.

This scheme thus offers a way to create programs for CP/M systems with many overlays that nevertheless consist of only two files: all of the program code and overlays are contained in a single PRG file.

Appendix G - The Overlay Loader

It may occasionally be necessary to write your own overlay loader to create programs that run under a non-CP/M compatible operating system, or to be able to do memory-mapped overlays, for instance. This appendix gives the technical information necessary to accomplish this task.

When a program contains overlays, Plink-II constructs a common block containing various tables for use by the overlay loader. This segment is named .OVTB., and is formatted as follows:



The overlay loader module must be in a file of the proper name (see section on selecting overlay loader), and must have a module name of .OVLY. It is automatically included in the program whenever a program contains overlays. When the program first begins execution, control is passed to the front of the overlay loader, which should initialize itself and then jump to label .INIT. The code here causes resident portions of the program which were not loaded by the operating system to be loaded by calling the .LOAD. entry point of the overlay loader with the overlay number in the A register. Execution of the actual program then begins via a jump to label .MAIN.

The overlaid symbol vectors are routines that are used as the program executes. Their task is to call the overlay loader to insure that an overlay is in memory:

```
CALL    .OVLY.
.BYTE   <overlay #>
.WORD   <symbol address>
```

When the overlay loader is called at its .OVLY. entry point, it must load the given overlay (if not loaded already), and then jump to the given address. It must save and restore all user registers, and it is probably a good idea to use the program's stack as little as possible to avoid an overflow.

The symbol vectors are used whenever an overlaid symbol is called from a section that is not a descendant of or the same as the section containing the symbol. In other words, the user's code is not modified when a reference is made from a section to its ancestor. This allows overlays to access data in their ancestors. Therefore, the overlay loader must guarantee that when an overlay is in memory, all of its ancestors are in memory also.

The overlay table contains the addresses needed to read the overlays from the .PRG file and ancestor information. It consists of a seven byte entry for each overlay, and the entries are implicitly numbered beginning with one. There are a maximum of 255 entries, each having the following format:

Offset	Size	Contents
0	1	Number of ancestor overlay, zero if none.
1	2	Starting memory address
3	2	Ending memory address
5	2	Starting disk record (128 bytes/record)

The table is ended by a dummy entry with a -1 (255H) in the ancestor field. Note that there is no ending disk record number: the overlay loader must check the starting disk address of the following entry to determine the disk size of the overlay. This size is different from the memory size, because the disk size is synchronized to a record boundry, and may be smaller than the memory size if there is uninitialized space at the end of the overlay. The last dummy entry in the table contains the record number of the end of the last overlay.

The algorithm to read an overlay into memory might proceed as follows. The overlay loader maintains an FCB for the .PRG file (its name may be obtained from address 50 in memory during the initialization phase). This file is positioned to the starting record number either by calculating an extent and FCB record number or by using any random I/O capabilities offered by the operating system. Then disk reads are done until either the end of the overlay is reached on disk or until memory space runs out. I/O's may be done directly into the user's memory, except that the last one may not have a full 128 bytes of memory and will have to be done in a separate buffer.

The overlay loader should keep track of what overlays are currently in memory (via a bit map, for instance), so that the loading process can be skipped if the overlay is already present. When an overlay is loaded, the loader should then delete any other overlays overlapping it in memory by comparing the starting and ending memory addresses.

The remaining task faced by the loader is satisfying the rule described above that an overlay's ancestors must be in memory when it is. If the applications to be handled never use overlay structures more than one level deep, then none of the overlays have overlays for ancestors, so nothing need be done. Otherwise, the ancestor requirement can be handled via the following two steps.

First, whenever an overlay is loaded, make sure its ancestor is loaded as well, and so on, until an overlay is encountered that is already loaded, or until the end of the ancestor chain is reached (ancestor number equal to zero).

Second, whenever an overlay is deleted, mark all of its descendants deleted also. Due to the fact that the overlay table is constructed so that an overlay always has a number higher than its ancestor, this process can be combined with the overlap check described above. The loader loops through the overlay table from start to finish. Each overlay is first deleted if it has an ancestor that's deleted; otherwise, the check for overlap with the loaded overlay is made. The ordering of the table guarantees that an overlay is deleted before any of its decendants are encountered.

Appendix H - Command Syntax Summary

This appendix defines Plink-II input syntax via a modified BNF for those familiar with formal computer languages. The symbol "::=" means "is defined to be". Angle brackets delimit names of subsequent definitions. Everything else must be entered as shown, except that key words may be abbreviated by truncating the lower case letters from the right. In any case, Plink-II translates any lower case letters in key words to upper case automatically. A vertical bar means choice: either of the items may be used. Sometimes, a set of alternatives is surrounded by parenthesis. Square brackets delimit optional input. Curly braces indicate input which may be omitted, or included as often as needed.

```
<input>      ::= <command> {;<command>}
<command>    ::= <statement> {<statement}>
<statement> ::=
   | OUtput <file>
   | Report [=<file>]
   | (IDent | MAIn) <module>
   | PROGid <program> [,<version>]
   | PDos <version>
   | MAP <option> {,<option>}
   | PAge <byte>
   | DAte
   | DEFine <definition> {,<definition>}
   | (FIle | LIbrary | SEArch) <file> {,<file>}
   | (CONcatenate | SEGment) <segment> {,<segment>}
   | (MOdule | INClude | EXClude) <module> {,<module>}
   | (LOcate|ACtual) <expr>
   | SECtion <id>6
   | SCtend
   | FReememory
   | BEginarea
   | SHare
   | OVerlay
   | ENDarea
   | Nosort
   | .DATA.
   | (DEBug | I8080)
   | DIsplay <s-item> {,<s-item>}
   | (GETVal | GETChr | PROMpt) <local> [,CR]
   | <local> := <expr>
   | [IF <relation>] (GOto | BAckto) <label>
<label> ::= <id>8 | <local>
<map option> ::= A | S | G | M
<definition> ::= <symbol> = <expr>
              |   <symbol> = <symbol> [(+|-) <expr>]
<s-item> ::= <literal> | TAB | CR | BEL | [HEX|DECimal] <expr>
<version> ::= <byte> [,<byte>]
<byte> ::= <expr> with msb=FF or 00
<relation> ::= <expr> <r-op>[<r-op>] <expr>
<r-op>       ::= < | > | =
```

```
<expr> ::= <value> [(+ | - | * | / | \ | & | !) <value>]
         | - <value>
<value> ::= <local> | <number> | <literal>2
<local> ::= #<value>, where value <= 50.
<program> ::= <id>6
<module>  ::= <id>6
<segment> ::= ["]<id>8
<symbol>  ::= <id>8
<id> ::= any string not beginning with a digit and
         not containing the following chars:
        ^=;<>/,\!'#&*+-:@ DEL
<file> ::= standard file name for operating system and
           not containing the chars:  ^?=_;<>/, DEL
<literal> ::= Any character string delimited by single
              quotes.  A quote character is included
              in a literal by entering two quotes.
<number> ::= Any string beginning with a digit 0-9
```

Appendix I - Error Messages

1 - "@" inside @-file. Disk files containing commands and used via an "@" may not contain further "@" specifications.

2 - "@" file is missing. The file name after an "@" specification doesn't exist.

5 - Input token too large. The string of characters entered at this point is too large to be a keyword, name, or literal.

6 - Invalid digit in number. Which digits are legal depends, of course, on the radix being used (default is hex).

7 - Number too large. The given value can't be expressed as a 16 bit quantity.

10 - Invalid file name. The input stream should contain a valid file name for the particular operating system being used.

11 - Expecting a statement. A key word which begins a statement should be present here.

12 - Expecting ":=".

13 - Invalid output file type. The type of the file given in the OUTPUT statement must be .COM or .PRG.

14 - Expecting identifier. A section, module, segment, or symbol name must be entered at this point.

15 - Expecting "="

16 - Expecting a value. An expression or 16-bit quantity must appear at this point.

18 - Data segments of modules may not be used in the CONCATENATE statement, because they are unique to a particular module.

19 - Expecting segment name.

20 - Expecting byte value. The value used here must lie in the range 0 to 255 (or -128 to 127 in signed notation). That is, it must be possible to represent the value in 8 bits.

21 - The IF <relation> statement must be followed by a GOTO or BACKTO statement.

22 - The label used in a GOTO or BACKTO statement does not exist.

23 - Invalid relational operator.

24 - Invalid DISPLAY item.

25 - expecting CR in GETCHR, PROMPT, or GETVAL statement.

26 - invalid local variable name. A local variable name is a pound sign "#" followed by a value in the range 0-50.

27 - Inconsistant statements for OUTPUT file type. Statements were used that imply a particular output file type, but a different type was specified. For instance, the OVERLAY statement may be used only with a .PRG file

40 - No more memory. Plink-II ran out of memory for its internal tables and disk buffers; therefore, the program could not be linked.

45 - Premature end of file. The end of the indicated .REL file was reached unexpectedly. Possibly, the .REL file was truncated by copying it with a program that assumes a CNTL-Z (1AH) is end of file. Try re-compiling.

46 - No disk directory space for output file.

47 - No more disk space for output file.

48 - Protection violation in output file.

49 - Can't close output file. Was the floppy diskette changed before Plink-II finished execution?

50 - Attempt to load to an absolute address. This error will occur if an attempt is made to .LOC to an absolute address in an assembler module and then load code or data. Use the LOCATE statement in Plink-II instead, making a common block out of the data that is to be loaded at a particular address.

55 - A segment was mentioned in the MODULE, SEGMENT or CONCATENATE statements, but was never encountered in the input .REL files.

60 - Undefined symbols exist. The listed symbols were external to one or more modules, but were never defined. They will have to be defined as internals of some module, or created via the DEFINE statement.

61 - Symbol is self-defined. The given symbol was defined relative to a another symbol, and that symbol defined relative to yet another symbol, and so on, until the original symbol was reached again. Thus, a circular chain was created with no symbol actually ever being defined.

65 - The .ROOT. section may not be an overlay.

66 - Overlays are nested too deeply. The maximum nesting level is 32 (i.e no more than 32 BEGINAREA statements may be used without entering an ENDAREA statement).

69 - Too many ENDAREA statements. The number of these must be equal to the number of BEGINAREA statements.

68 - Not enough ENDAREA statements. The number of these must be equal to the number of BEGINAREA statements.

69 - Too many overlays or overlay entry points. There may be a maximum of 255 overlays and about 9000 overlay entry points.

70 - LOCATE error. The specified address overlaps previously loaded items.

71 - Program address space is too large. The program will not fit into the address space of a 16-bit computer.

72 - Either no modules were included in the linkage edit, or the entire program is overlaid. At least one section of the program must be permanently resident, and must contain something.

73 - loading below address 100H in .COM file. Since .COM files are always loaded at address 100H by the operating system, nothing may be loaded below this address.

74 - The OVERLAY and OVERLAP statements may be used only inside an overlay area (BEGINAREA/ENDAREA).

78 - No FILE, LIBRARY or SEARCH statement has been given yet, so INCLUDE and EXCLUDE may not be used.

79 - Can't INCLUDE and EXCLUDE from the same file. Only one of these statements may be used with a particular file.

80 - Duplicate input file. Each file used in the FILE, SEARCH or LIBRARY statements must have a unique name.

83 - The program starting address defined by a particular module may not be an absolute address, or a symbol external to that module.

84 - The module mentioned in the MAIN statement was never encountered in the input .REL files.

85 - Missing starting address. The MAIN statement was not used, symbol .MAIN. did not exist, and none of the program modules had a defined starting address.

86 - .MAIN. may not be an absolute symbol, or an external symbol. It must be an INTERNAL symbol of some module.

91 - Invalid external byte reference. In the PSA macro assembler module named, an external byte reference (whose relocation base number is supplied) was made to a non-absolute symbol, or one with too large a value. External byte references may be made only to absolute symbols with a value in the range 0 to 255 (or -128 to 127 in signed notation).

93 - MicroSoft Commons overlap in memory. Due to the defined overlay structure, two or more MicroSoft common blocks that are accessed by the same module overlap in memory. This is illegal because it makes "fix ups" to already loaded code impossible.

94 - Too many common blocks in MicroSoft module. A maximum of 251 common blocks are allowed in each module.

95 - Invalid special link item in MicroSoft file. This .REL file is either smashed, or is not legally formed according to the MicroSoft format, or represents a MicroSoft Cobol program containing SECTION statements.

96 - Invalid B field in MicroSoft .REL file. This field is supposed to contain an ascii name of up to 8 characters. The .REL file is probably smashed: try re-compiling.

97 - More than one library search was called for in a MicroSoft format module that is itself being library searched. This MicroSoft format module was incorrectly generated and may not be linked.

100 - expecting PSA .REL module. This error often occurs when there is garbage at the end of the previous module in a library file.

101 - Invalid record type in PSA .REL file. The file is probably smashed: try re-assembling.

102 - Expecting carriage return (ODH) in PSA .REL file.

103 - Expecting line-feed (OAH) in PSA .REL file.

104 - Expecting identifier in PSA .REL file.

Diagnostic Errors

Errors with the following numbers indicate that a bug in Plink-II has occurred through no error on the user's part. Try running Plink-II again. If the error persists, please gather the relevant information (error message, Plink-II version, input files, etc.) and notify the sofware distributer from whom you obtained Plink-II. These error messages are documented only for completeness in this manual.

150 - Too many files open simultaneously.

151 - File de-allocation bug.

155 - Relocation base is undefined.

156 - Relocation base undefined for disk address.

157 - Bad relocation base (BS.SCT)

160 - Duplicate segment (SG.ADD).

161 - Missing segment (SG.FND).

162 - End of segment list (SG.FST, SG.NXT).

163 - Segment not in assigned Section's segment list (SGsect).

164 - Segment undefined (SG.OFF).

170 - Missing symbol.

171 - End of symbol list.

172 - Undefined symbol.

173 - Self-defined symbol.

175 - End of Section list.

180 - End of File list.

181 - Module excluded/not included.

185 - End of PSA .REL module.

190 - MicroSoft fixup bug (FixLnk)

191 - MicroSoft Cobol symbol bug (ModEnd).

250 - Aborted command.

1.08
PATCH FOR BASIC
TLINK-II.COM
3948 21 → 39
USE .DATA.
BASCOM —— /O

1.10A
PATCH FOR BASCOM 5.3 USERS
33DA ~~1~~
30 → 18
3697
23 → C9
NO OVERLAYS

Update Notes for Version 1.10

This note lists Plink-II bug fixes and enhancements effective with version 1.10.

Bugs:

- Some versions of the Digital Research PL/1 compiler generate invalid .REL files when an empty string is declared: 256 garbage bytes are written to the .DATA. segment causing the linkage editor to overwrite the memory assigned to the following segment. This usually doesn't matter since the next segment is typically from a module that hasn't been loaded yet, but the problem may surface if some of Plink-II's commands for moving segments around in the program are used (e.g. SEGMENT, MODULE, LOCATE). Another PL/1 problem has been fixed: modules having no code in them are now handled correctly. These would sometimes cause fixup errors or missing code or data in the output file.

- If more than 2 undefined segments exist (warning #55), Plink-II would die with diagnostic error #164. Now programs may be linked with undefined segments: they are ignored. This is useful for tasks such as the creation of overlay structures that are used in many programs where not all of the segments mentioned in the structure are used in each program.

- DEFINE <symbol> = <local variable> would cause a syntax error. This has been corrected.

- Common blocks would not be handled properly if they had the same name as a module: each instance of the common was assigned a different memory area. This situation is now handled properly.

  MicroSoft apparently handled this problem in their Fortran by setting the high order bit in the first character of each common block name. Unfortunately, this means that MicroSoft Fortran and Basic modules may not be combined into the same program because the new Basic compiler does not set the bit. Plink-II can match the common names properly, but another Basic problem is described below.

- The overlay loader would be incorrectly linked in a .COM program when the LOCATE command was used. This has been fixed.

New Features:

- When the LOCATE statement is used in a .PRG program a new section is created. Since only the main section is loaded by the operating system when the program is executed, Plink-II includes the overlay loader into the program to load the other sections before execution of the program begins. This action may now be inihibited by using the new PAD option provided with the LOCATE command. For example,

  LOCATE 3000, PAD

  causes the previous section to be filled out until it reaches address 3000. In this way, the next section is forced to be adjascent to the previous one and will be loaded at the same time

from disk: the overlay loader is not required.

- The COMMON command has been added. It may be used to define the size of a common block to be larger than the definition of the common provided by any module. It also performs the function of the SEG command by placing the common block in the current section. The syntax is COMMON <name1> = <size1>, <name2> = <size2>, ... For example:

  COMMON C1=100, C2 = #1 * 5

  The COMMON command may not be used on a .DATA. segment or error #18 will result. Also, the COMMON and CONCATENATE statements should not be used on the same common or the results will be unpredictable.

- The various overlay loaders selected via the DEBUG and I8080 commands are now combined into a single library file called OVERLAY.REL instead of being in separate files. Each has a different module name: .OVLZ. is the standard loader, .OVLD. is selected when the DEBUG command is used, and .OVL8. when the I8080 command is used. Older overlay .REL files should be discarded: they will no longer work.

- Many people have reported problems with linking modules produced by MicroSoft's new Basic compiler. Plink-II can now handle these. If the /O switch is used in the Basic compiler (to avoid the use of the runtime module) some of Plink-II's special commands have to be used.

  The problem is that the initialization code in these Basic programs attempts to clear the blank common (.BLNK.) and counts on having the data segment for the module ("BASIC) immediately following .BLNK. in memory in order to determine its ending address. Plink-II normally moves .BLNK. to the end of the section so that Fortran programs can access free memory; therefore, the clearing routine wipes out the operating system.

  The SEGMENT command of Plink-II can be used to specify that the named segments are to be allocated memory at the current load address. Older versions of Plink-II always put .BLNK. at the end of the current section, but version 1.10 will not do this if .BLNK. is used in the SEGMENT command. Plink-II will move all uninitialized segments to the end of the section, and .BLNK. and "BASIC are normally uninitialized, so the NOSORT command must also be used to inhibit this action. To sum up, entering

  SEGMENT .BLNK., "BASIC NOSORT

  into the Plink-II command will cause the correct memory structure to be be generated.

  If the /O switch is NOT used in the Basic compiler, the program is linked so as to use the runtime support module, and no extra Plink-II commands are required: everything is sorted into the correct order in memory. A .COM file must be specified for output, overlays may not be used, and the LOCATE, ACTUAL, and .DATA. commands should not be used.

Valid Plink input files:

The MicroSoft relocatable file format has become an industry standard for Z80 - CP/M compiler output. However, several manufacturers are selling compilers which output files that look like MicroSoft's, but actually contain subtle differences. Plink can handle some of these, but others will not work. Also, MicroSoft periodically makes minor changes to the format in order to support new language features.

The list below indicates which compilers have been checked out with Plink 3.28. Usually, lower numbered versions are also handled correctly. If a compiler you wish to use is not on this list, either insure that it outputs a format compatible with a listed one, or contact your software distributor.

MicroSoft:
- Cobol 4.01 (in 4.01, the SECTION statement may not be used).
- Fortran 3.31
- Basic 5.30 (see comments above).

Cromemco:
- Cobol 3.01

Digital Research
- PL/1 1.3 (the indexed .IRL files are not supported: they will have to be converted to normal form with the LIB program).

MT MicroSystems:
- Pascal MT+ 5.2 (.ERL files designed to be input to the disassembler are not supported).

Whitesmiths
- C 2.0 (see discussion in update letter for Plink-II 1.08)

Ithaca Intersystems
- Pascal/Z 3.0.

- Output: .COM file or PSA .PRG file with overlays. The .PRG files execute directly only under the PDOS operating system, but a utility program is supplied to allow them to be run in a CP/M type environment. Dual 1K output buffers insure adequate execution speed, and any random I/O calls supported by the operating system are used.

- Memory Map: Plink-II can create 4 different reports: symbols sorted alphabetically, program segments sorted alphabetically, the entire program printed in address order, and program modules printed with date and time of creation and version number (PSA .REL files only). The reports may be written to the list device or a disk file. Variable page width is supported. The date and time is included (on systems not supporting date and time, blanks are used, or Plink-II can prompt the operator).

- Common Blocks: Plink-II supports Fortran type common blocks, with no requirement that the largest instance of the common be input first. Also, "concatenated" common blocks may be created: in these commons, each module is allocated its own space in the common, instead of sharing the space with other modules. This feature is useful for tasks such as creating a table where each entry in the table is contributed by a separate module.

- Memory allocation: program modules are broken down into code, data, and common segments, re-grouped into "sections", and allocated at any memory address desired by the user. A symbol .END. is created to signify the free memory available to the program, and its location may be defined at any point. The free memory symbols used by the MicroSoft and Whitesmith formats are handled as well. Program sections may be set up to execute at an address different than the load address.

- Library Search: Libraries of relocatable files may be loaded in their entirety, or with specific modules included or excluded. They may be searched, so that a module is included only if required by portions of the program already loaded. Each library file may be individually marked to be searched only once, or to be searched repetitively until all references to undefined symbols have been satisfied. Library modules may be inserted enmasse into the main program or a particular overlay, or may be separately allocated by the user to different program sections.

- Overlays: any number of "tree-structured" overlay areas may be created, to a maximum of 255 overlays and to a depth of 32 levels, and these may be overlapped in an arbitrary fashion to create non-tree-structured organizations. Each overlay can have any number of entry points. An overlay loader module is automatically included with the program. When an overlay entry symbol is accessed, the overlay and all of its "ancestors" in the overlay tree are automatically loaded. No source changes are required in the program modules. The main program and all overlays are created with a single linkage edit, and output to a single file. Memory overhead is approximately 7 bytes per overlay, 6 bytes per overlay entry symbol and 1K bytes for the overlay handler. Output file disk space may be minimized by having Plink-II "sort" the uninitialized portions of each overlay to the end: these portions need not appear in the disk file. Versions of the overlay loader which run on an 8080 and display debugging messages as the program executes are offered. Technical information allowing user-created overlay loaders is supplied.

- Control Statements: Fifty temporary 16 bit variables are supplied. These may be assigned values, simple arithmetic and logical expressions, or console input, and may be tested via relational operators and branch instructions. Temporary variables, character strings, and simple expressions may be displayed on the console.

Plink-II is a part of Phoenix Software Associates' systems software development package, Pdevelop-II.