HDY

North American Response Center

HP 3000 APPLICATION NOTE #34

Extra Data Segments

(Using COBOLII Examples)

HEWLETT PACKARD

November 1, 1987
Document P/N 5958-5824R2745

RESPONSE CENTER APPLICATION NOTES

HP 3000 APPLICATION NOTES *are published by the North American Response Center twice a month and are distributed with the Software Status Bulletin. These notes address topics where the volume of calls received at the Center indicates a need for addition to or consolidation of information available through HP support services.*

Following this publication you will find a list of previously published notes and a Reader Comment Sheet. You may use the Reader Comment Sheet to comment on the note, suggest improvements or future topics, or to order back issues. We encourage you to return this form; we'd like to hear from you.

USING EXTRA DATA SEGMENTS

The purpose of this document is to give experienced programmers information on using Extra Data Segments on the HP 3000. A COBOLII example is provided to illustrate the techniques involved.

Overview

There are numerous applications for Extra Data Segments. Such applications include providing a means for processes in a process tree to communicate or storing large arrays that cannot fit in the program's stack due to their size. Although, of course, no data segment (regardless of type) may exceed 32,768 words (65K bytes), programs that need to use very large arrays can 'offload' these structures into an XDS.

Referencing data in an XDS

The two types of data segments directly accessible to users of MPE are *stacks* and *extra data segments.*

Data in a stack is referenced by MPE through the use of pre-defined registers: the DL, DB, Q, S, and Z. These are described further in *HP 3000 Stack Operation, Appnote #6, May 15, 1986.*

Unlike a stack, an Extra Data Segment or XDS, is not referenced by a series of registers. Instead, it is considered unformatted because it is up to the programmer to decide upon how it will be used and what it will contain.

The following terms will be used in this document to refer to entries in the XDS:

Logical Entry	This is analagous to a record in a disc file. It is the definition of one entry contained in the XDS. It may be no smaller than 1 word (2 bytes) and must be an integral number of words no larger than the XDS itself.
Displacement	This is the offset from 0 to the beginning of the logical entry to be read or written to. Using the disc file analogy, this would be the relative record number.
Entry Length	This is the number of words (not bytes) that make up one logical entry. This also defines the number of words to be transferred to or from the XDS. It may be no less than 1 word and no greater than the size of the XDS.

System Intrinsics

The following System Intrinsics are used to create and access an Extra Data Segment:

ALTDSEG	Alters the size, in 4 word allotments, of existing extra data segments.
DMOVIN	Transfers a specified number of words from an extra data segment to a program's stack.
DMOVOUT	Transfers a specified number of words to an extra data segment from a program's stack.
FREEDSEG	Releases an extra data segment, thereby erasing it from memory.

GETDSEG	Creates an extra data segment or grants access to a sharable extra data segment.

Extra Data Segment or DS Capability

In order to use these intrinsics a program must be PREPed with the DS capability. Additionally, the group and account in which this program will reside must also have DS capability, although users of such programs DO NOT need to possess DS.

There are two types of extra data segments: *sharable*, those that may be accessed by any program knowing its 'ID' in the current job or session and *private*, those known only to the process that created them.

Both sharable and private extra data segments are accessed in the same way. A sharable XDS is analagous to a temporary file in that it remains associated with the job or session under which it was created and is purged automatically when this job or session logs off. A private XDS, however, is purged when the process or program which created it completes.

An Example Using Extra Data Segments

For the rest of this discussion please refer to the following excerpt from a COBOLII program:

```
WORKING-STORAGE SECTION.
01  INTEGER-MASK                PIC ZZZZ9.
01  XDS-CONTROL-DATA.
    05  XDS-INDEX               PIC S9(4) COMP.
    05  XDS-ID                  PIC S9(4) COMP.
    05  XDS-NAME REDEFINES XDS-ID PIC X(2).
    05  XDS-LENGTH              PIC S9(4) COMP.
    05  XDS-LOGICAL-ENTRIES     PIC S9(4) COMP VALUE 100.
    05  XDS-ENTRY-LENGTH        PIC S9(4) COMP VALUE 40.
    05  XDS-DISPLACEMENT        PIC S9(4) COMP.
    05  XDS-NUM-LOADED          PIC S9(4) COMP.

01  XDS-BUFFER.
    05  XDS-CUSTOMER-NAME       PIC X(30).
    05  XDS-ADDRESS             PIC X(30).
    05  XDS-AR-AMOUNT           PIC S9(9)V99 COMP-3.
    05  XDS-AR-BILL-DATE.
        10  XDS-AR-BD-YY        PIC X(2).
        10  XDS-AR-DB-MM        PIC X(2).
        10  XDS-AR-BD-DD        PIC X(2).
    05  XDS-NUM-TRANS           PIC S9(9) COMP.
    05  FILLER                  PIC X(4).
```

Creating and Accessing an XDS

In this example, each logical entry will be 40 words in length and a segment large enough to hold 100 of these entries will need to be created. The following calculation is used to determine the size of the XDS:

```
XDS-LENGTH = XDS-LOGICAL-ENTRIES * XDS-ENTRY-LENGTH.
```

This results in a request for an XDS of 4000 words.

Notice that the item XDS-BUFFER is made up of several different data types. How your program interprets the data in the XDS is limited only by the data types supported by the language used to code the program.

Shared or Private XDS

To designate that an XDS is to be private (non-sharable), a zero XDS-ID value must be passed to GETDSEG otherwise the XDS will be considered to be sharable by any process running under the job or session that created it.

```
MOVE 'D1' TO XDS-NAME.
```

For instance, here, the XDS will be sharable and will be called 'D1', this name is arbitrary and may be any value such that XDS-ID, the 'ID' passed to GETDSEG, is non-zero.

Now GETDSEG is called to have MPE allocate a block of memory and attach this block to the program requesting it:

```
CALL INTRINSIC "GETDSEG" USING XDS-INDEX,
                               XDS-LENGTH,
                               XDS-ID.
IF C-C < 0 THEN
     .
     .
     .
```

The condition code returned is checked for a 'less than zero' value to see if the the call has failed. If a condition code greater than zero is returned, you have been granted access to an existing XDS. It is important to remember that a shared XDS will survive the process that created it and if the process does not explicitly delete the data segment with the FREEDSEG intrinsic the XDS will remain. Other processes run from the session or job under which the XDS was created would use the same XDS if the same XDS name were used.

If this call is successful then XDS-INDEX will contain a value that the other data segment management intrinsics can use to locate this XDS. This value MAY NOT subsequently be changed by your program.

Transferring Data to an XDS

Now that an XDS has been created it is possible to load data into it. To do this you will have to know what the 'displacement' into the XDS will be. This value is ALWAYS calculated from zero since the first word of an XDS is considered the zeroth word. To calculate the displacement:

```
COMPUTE XDS-DISPLACEMENT = XDS-ENTRY-LENGTH * XDS-NUM-LOADED.
```

The item XDS-NUM-LOADED is used to count the number of entries that have already been placed into the XDS. On the first iteration this value would be zero which would then result in XDS-DISPLACEMENT being zero. On the next iteration XDS-NUM-LOADED would be 1 so the displacement would be 40, remember the first entry occupies words 0 thru 39!

Once the displacement has been calculated the DMOVOUT intrinsic is called to cause the transfer to occur:

```
CALL INTRINSIC "DMOVOUT" USING XDS-INDEX, XDS-DISPLACEMENT,
                               XDS-ENTRY-LEN, XDS-BUFFER.
IF C-C NOT = 0
      .
      .
      .
```

Transferring Data from an XDS

Transfering data into WORKING-STORAGE from an XDS is similar to transferring data to an XDS, a displacement must be calculated using the number of logical entries and the length of this entry to calculate a displacement from word zero of the XDS. Once this is done the DMOVIN intrinsic is used to initiate the transfer:

```
CALL INTRINSIC "DMOVIN" USING XDS-INDEX, XDS-DISPLACEMENT,
                              XDS-ENTRY-LEN, XDS-BUFFER.
IF C-C NOT = 0
      .
      .
      .
```

This will cause 40 words (the value of XDS-ENTRY-LEN) of data beginning at the word in the XDS pointed to by XDS-DISPLACEMENT to be transferred into XDS-BUFFER.

Any time data is transferred into your programs stack the ONLY limitation on placement of that data are the absolute bounds of the stack, e.g. the DL and Z registers. (The DL, or data limit register, points to the absolute boundary of user accessible data. The Z, or stack limit register, points to the absolute upper boundary of the stack.) Since MPE is not aware of data definitions within a stack it will transfer to ANY requested location so long as it does not go below DL nor above Z.

If such a location is determined incorrectly, this can cause great problems depending upon what portion of the stack is overwritten when the data is transferred from the XDS. At a minimum, random data corruption can occur. At worst, either CST or STT violations may occur because portions of the STACK MARKER or locations used to store COBOL PERFORM return addresses could be overwritten. If a program is running in PRIVILEDGED MODE, where no bounds checking is done at all, the results can be even more disastrous. Therefore, before using such techniques be sure you understand their implications and have taken the necessary steps to prevent accidents.

Altering the Size of an XDS

When an XDS is initially created, space in Virtual Memory (on disc) is reserved for it when it is not needed in main memory. Virtual Memory is allocated in 'pages' of 512 words which corresponds to 4 disc sectors (128 words each). In this example an XDS of 4000 words was requested however space was set aside in Virtual Memory for 4096 words (8 pages of virtual memory, 512 words each).

The ALTDSEG intrinsic will allow an XDS to be 'sized' up or down in a minimum of 4 word allotments. A request to increase the size of an XDS can never exceed the initial space requested (in this example, 4096 words). An attempt to request more space than is available will result in the condition code (C-C) being set greater than zero:

```
CALL INTRINSIC "ALTDSEG" USING XDS-INDEX, 100, XDS-SIZE.
IF C-C > 0 THEN
    MOVE XDS-SIZE TO INTEGER-MASK
    DISPLAY "REQUESTED SIZE EXCEEDS MAXIMUM AVAILABLE"
    DISPLAY "NEW SIZE SET TO MAXIMUM VALUE: ", INTEGER-MASK.
```

Here, 100 additional words were requested but only 96 remain so ALTDSEG sets C-C greater than zero to indicate this and the size of the XDS was increased to 4096 words, the maximum available.

WARNING

Sizing an XDS up or down will cause the XDS to be copied to Virtual Memory and then read back into 'main memory'. This transfer will cause your program to wait for its completion. For this reason ALTDSEG should be used infrequently, if at all. When an XDS is created enough space should be allocated initially to avoid the need to 'size' your XDS during its life cycle.

Erasing an XDS

An XDS may be erased, releasing the main and virtual memory it had occupied. Shared XDSes are deleted when the job or session that created them logs off. Private XDSes are deleted when the process (program) that created them completes.

To free memory occupied by an XDS during execution of the program that created it you would call FREEDSEG:

```
CALL INTRINSIC "FREEDSEG" USING XDS-INDEX, XDS-ID.
IF C-C < 0 THEN
      .
      .
      .
```

If the C-C returned is 'LESS THAN' zero, FREEDSEG has failed. A CCE (C-C = 0) returned indicates the XDS was deleted while a CCG (C-C > 0) indicates that this program's access to it has been removed but the XDS sill exists and is being shared by other processes in the 'process tree'.

Extra Data Segment System Configuration Considerations

The following system configuration values should be considered if you intend to make use of extra data segments in your software applications:

```
MAX EXTRA DATA SEGMENT SIZE = <XXXXX> (MIN=0, MAX=32764)

MAX # OF EXTRA DATA SEGMENTS/PROCESS = <XXX> (MIN=0, MAX=255)
```

These values are set during the SEGMENT LIMIT section of the SYSDUMP dialog.

In addition you should also consider altering the value of the SWAP TABLE. This table is used to store a process' 'locality' which is a list of the code and data segments needed in memory in order for the process to run. If this table is underconfigured a System Failure 602 will result.

Lastly, the amount of virtual memory allocaton configured for your system should be evaluated if extra data segments are to be used. Since each extra data segment (in fact all data segments) will occupy a space in virtual memory at least as large as the requested size of the segment in main memory. On MPE V systems virtual memory can be allocated on all drives except LDEV 1 using either a COOLSTART or COLDSTART. A RELOAD is necessary to change the virtual allocation on logical device 1.

BACK ISSUE INFORMATION

Following is a list of the Application Notes published to date. If you would like to order single copies of back issues please use the *Reader Comment Sheet* attached and indicate the number(s) of the note(s) you need.

Note #	*Published*	*Topic*
1	2/21/85	*Printer Configuration Guide (superseded by note #4)*
2	10/15/85	*Terminal types for HP 3000 HPIB Computers (superseded by note #13)*
3	4/01/86	*Plotter Configuration Guide*
4	4/15/86	*Printer Configuration Guide - Revised*
5	5/01/86	*MPE System Logfile Record Formats*
6	5/15/86	*Stack Operation*
7	6/01/86	*COBOL II/3000 Programs: Tracing Illegal Data*
8	6/15/86	*KSAM Topics: COBOL's Index I/O; File Data Integrity*
9	7/01/86	*Port Failures, Terminal Hangs, TERMDSM*
10	7/15/86	*Serial Printers - Configuration, Cabling, Muxes*
11	8/01/86	*System Configuration or System Table Related Errors*
12	8/15/86	*Pascal/3000 - Using Dynamic Variables*
13	9/01/86	*Terminal Types for HP 3000 HPIB Computers - Revised*
14	9/15/86	*Laser Printers - A Software and Hardware Overview*
15	10/01/86	*FORTRAN Language Considerations - A Guide to Common Problems*
16	10/15/86	*IMAGE: Updating to TurboIMAGE & Improving Data Base Loads*
17	11/01/86	*Optimizing VPLUS Utilization*
18	11/15/86	*The Case of the Suspect Track for 792X Disc Drives*
19	12/01/86	*Stack Overflows: Causes & Cures for COBOL II Programs*
20	1/01/87	*Output Spooling*
21	1/15/87	*COBOLII and MPE Intrinsics*
22	2/15/87	*Asynchronous Modems*
23	3/01/87	*VFC Files*
24	3/15/87	*Private Volumes*
25	4/01/87	*TurboIMAGE: Transaction Logging*
26	4/15/87	*HP 2680A, 2688A Error Trailers*
27	5/01/87	*HPTrend: An Installation and Problem Solving Guide*
28	5/15/87	*The Startup State Configurator*
29	6/01/87	*A Programmer's Guide to VPLUS/3000*
30	6/15/87	*Disc Cache*
31	7/01/87	*Calling the CREATEPROCESS Intrinsic*
32	7/15/87	*Configuring Terminal Buffers*
33	8/15/87	*Printer Configuration Guide*
34	9/01/87	*RIN Management (Using COBOLII Examples)*
34	10/01/87	*Process Handling (Using COBOLII Examples)*
34	11/01/87	*Extra Data Segments (Using COBOLII Examples)*