Malcolm Bull
Training and Consultancy Publications

MB-Guide to System debugger

Malcolm Bull

MB-Guide
to
the System debugger

MB-Guide

to

the System debugger

by

Malcolm Bull

MALCOLM BULL Training and Consultancy Services

Introduction

This *MB-Guide to the system debugger* is produced for those who need a quick introduction to the features of the spooler on the Pick operating system.

This *MB-Guide* contains:

1) A general introduction to the system debugger

2) A description of the individual commands which are available for use with the debugger

3) A discussion of the use of the debugger in patching frames to recover from GFEs and other errors.

Throughout any treatment of the interactive system debugger, the reader should continually be aware of the potentially catastrophic damage which can be wrought by misuse of the tool. Whilst it could be said that the system debugger is the supreme example of *a little knowledge being a dangerous thing*, it is nevertheless a valuable tool for the experienced and competent user.

The guide is suitable for beginners, who need only read the first few sections (in which we consider the G / OFF / END and P commands) and who would be overawed by the rest, and for the technical user and the System Manager, who can use all the sections. It is not intended for use by Assembly language programmers; these would require a much more detailed explanation of the debugging facilities, such as breakpoints, than is given here.

You may find the following titles in the *MB-Guide* series useful in conjunction with the present volume:

Group format errors
The Basic symbolic debugger

This *MB-Guide* is not intended to present a complete description of the subject but merely to place it in context and give the reader enough information to use the facilities and to survive.

Best use can be made of this *MB-Guide* if it is read in conjunction with the reference literature which is provided for your system. You should amend your copy of this guide so that it accurately reflects the situation and the commands which are used on the implementation which you are using. By doing this, your *MB-Guide* will become a working document that you can use in your daily work.

I hope that you enjoy reading and using this *MB-Guide* and the others in the series. A list of the current titles is given at the end of this guide.

Malcolm Bull

ISBN: 1 873283 11 3

1 Introducing the debugger

The Basic symbolic debugger and the interactive system debugger are standard parts of the standard Pick system software and are provided to enable the programmer and the System Manager to locate, to identify and possibly to correct software errors.

Although they may be invoked directly by the user, by pressing the

<BREAK>

key and by other means which we shall discuss below, the debuggers are normally only encountered when a software error occurs.

2 End-users and the debugger

If I'm not a technical user and I just want to use the system to run programs which have been written by someone else, how much do I need to know about this debugger?

Very little. The non-technical user need only recognise that a display such as

```
*I 123
*
```

indicates that the Basic symbolic debugger has been invoked during the execution of a Basic program or subroutine. This will almost certainly because something has gone wrong with the program, possibly because the program has not been tested properly, and a message will probably be displayed saying what the problem is. Typical messages might look like this:

```
[B17] LINE 234 ARRAY SUBSCRIPT OUT-OF-RANGE
*I234
*
```

or

```
[B12] LINE 76 FILE HAS NOT BEEN OPENED
*I76
*
```

The interactive system debugger will display something like this:

```
I 123.45
!
```

to indicate that the debugger has been invoked during the execution of a standard system process - such as LIST or OFF verbs. Other than recognising these two situations, should they ever occur, the ordinary end-user need not concern himself/herself with either of the debuggers.

If you do accidentally hit the <BREAK> key or either of the above sequences is displayed for any other reason, then you should:

1) Make a note of any error messages which might be displayed,

2) Make a note of what you were doing and what the system seemed to be doing when the problem occurred,

3) Report the message to your supervisor, to the System Manager or to the analyst or programmer who is responsible for the system.

You have now got to decide whether you want to try and continue, if this is possible, or whether you want to abandon the process.

You could attempt to continue with the processing from the point where it was interrupted. To do this

4) type

   G

   and then press the <RETURN> key to attempt to continue the processing from the point at which it was interrupted.

   This will almost certainly work in those situations where you have accidentally pressed the <BREAK> key.

If you take this last step and this produces yet another error message or if it does not appear to succeed, then you should:

5) type

   END

   and then press the <RETURN> key to abandon the process, or you could

6) type

   OFF

   and then press the <RETURN> key to abandon the process and log off the account.

You should remember that, if you type END or OFF when the system was in the *middle* of its processing, it may be that your data has not been processed thoroughly or consistently. For this reason, you should make a note of the circumstances in which the process was abandoned, and check the data which you were processing at that time to ensure that the processing had completed properly.

You may find it useful to read the section on the General

Debugger Commands below.

3 Invoking the debugger

Under normal circumstances, it is possible to interrupt any process by pressing the <BREAK> key on the terminal keyboard. If you interrupt a Basic program whilst it is executing, then you will get a display something like:

```
*I40
*
```

showing that you have interrupted the program at line 40 and you are in the **Basic symbolic debugger**, as discussed in the *MB-GUIDE to the Basic symbolic debugger*.

If you interrupt any process other than a Basic program, such as an Access report or a Basic program compilation, you will get a display something like:

```
I 123.45
!
```

Showing that the process has stopped at the instruction at byte 45 (hexadecimal 45 = decimal 69) of frame number 123, and you are in the **interactive system debugger**.

The various means of invoking the interactive system debugger are:

1) By pressing the <BREAK> key.

2) By entering the

   DEBUG

   command from within the *Basic symbolic debugger.*

3) On Open Architecture and Advanced Pick, there are facilities for **level-pushing**, allowing one process to be interrupted and another one invoked). This facility can be invoked by either the <ESC> key or the <BREAK> key. On such implementations, there are facilities for specifying which key is to be used to invoke the debugger and which to invoke the level-pushing. There is a TCL

   DEBUG

   statement to invoke an interrupt.

4 Debugger commands - introduction

When either of the Pick debuggers is invoked, the system will display a

! when within the interactive system debugger, or

* when within the Basic symbolic debugger.

at this point, the user may enter any one of a number of commands. In this present beginner's guide, we are only concerned with those which relate to the interactive system debugger, and for the purposes of this discussion, we shall group them under the following headings:

1) general commands which are available to all users and which apply equally to both the Basic symbolic debugger and the interactive system debugger.

2) commands which are specific to the interactive system debugger and allow the user to display any parts of virtual memory.

3) commands which are specific to the interactive system debugger and allow the user to change any parts of virtual memory.

   This set of commands can be used to patch and/or remove group format errors.

4) commands which are specific to the interactive system debugger and assist the programmer in debugging Assembly language routines.

5) commands which are specific to the interactive system debugger and are used for decimal/hexadecimal conversion.

The commands in groups (2), (3) and (4) are only available to users with SYS2 privileges and can only be issued when the debugger is available. The interactive system debugger may be disabled from the SYSPROG account by means of the DB command described below.

The following commands are discussed below:

<LINE FEED>	L
<RETURN>	M
A	ME
ADDD	MULD
ADDX	MULX
B	N
C	off<LINE FEED>
D	off<RETURN>
DB	P
DIVD	SUBD
DIVX	SUBX
DTX	T
E	U
end<LINE FEED>	X
end<RETURN>	XTD
G	Y
K	Z

4.1 Debugger commands - general

The first group of commands are available with both the interactive system debugger (at the ! prompt) and the Basic Symbolic Debugger (at the * prompt).

<RETURN>
: used on its own, the <RETURN> key has no effect, except to repeat the ! or * prompt.

In general, <RETURN> after any of the debug commands which we discuss in this *MB-Guide* will process the command but stay in the debug mode.

<LINE FEED>
: will leave the debugger and continue the processing, if possible. This is identical to the G command described below. The down-arrow cursor control key also has the same effect.

P
: will switch the terminal print output function ON or OFF, suppressing the display to the terminal screen, and will remain in the debugger.

  This is useful if you want to stop the display which is being produced but you do not want to stop the processing. This will usually speed up the time taken to execute.

end<RETURN>
END<RETURN>
: will terminate the processing and return to TCL, unless attribute 9 of your account definition item contains the letter R and thereby prevents you from doing this, in which case you will be returned to the logon Proc.

end<LINE FEED>
END<LINE FEED>
: will terminate the current activity and return the processing to the calling Proc, if the current activity was invoked from a Proc, otherwise the processing will return to TCL.

off<RETURN>
off<LINE FEED>
OFF<RETURN>
OFF<LINE FEED>
: will terminate the current activity and log off the account.

G
: will continue the processing, if possible. This is identical to the <LINE FEED> command described above. The G command is useful if you are using a terminal with no <LINE FEED> key.

The interactive system debugger will accept these commands in upper-case or lower-case, as shown. The Basic symbolic debugger will only accept commands in upper-case.

4.2 Interactive system debugger commands - summary

The above commands are available with both the interactive system debugger (at the ! prompt) and the Basic symbolic debugger (at the * prompt).

The following commands are only acceptable to the interactive system debugger and may only issued by users whose account has SYS2 privileges and can only be issued when the debugger is available.

A

will display the address at which the execution was interrupted. This will normally be the same as the address on the

```
! fid.ddd
```

display.

Caddress;window

will display the contents of the virtual memory (the number of bytes is specified by **window**) starting at position specified by **address**, in **character** format, and allow the user to amend the data.

Lfid

will display the link fields of frame specified by fid. For example:

```
!L12345<RETURN> 7 :_ 12346 12344 :_ 2
```

indicating that there are 7 following contiguous frames starting at forward link 12346, and 2 previous contiguous frames starting at backward link 12344.

The TCL DUMP command also has options to display the linkage information:

```
DUMP 12345 L
```

will display the linkage details on all the contiguous frames following frame 12345, and

```
DUMP 12345 LU
```

will display the linkage details on all the contiguous frames preceding frame 12345.

Xaddress;window

will display the contents of the virtual memory (the number of bytes is specified by **window**) starting at position specified by **address**, in **hexadecimal** format, and allow the user to amend the data.

In addition to the Caddress;window command (to display the data in character format) and the Xaddress;window command (to display the data in hexadecimal format), there is an Iaddress;window command (to display the data as an integer number). In general the C and X forms are easier to use and we shall not consider the I form here.

4.3 Address

Interactive system debugger commands such as

```
Caddress;window
Xaddress;window
```

require you to specify the location of a part of the virtual memory (disk) storage which is to be inspected and/or changed.

On such commands, the location of the data within virtual memory - its **address** - is specified by means of two pieces of data: the frame identifier (the FID), and the **displacement.**

The FID of the frame may be expressed as a decimal number:

```
4660
```

to identify frame 4660 of the virtual memory storage, or as a hexadecimal number:

```
.1234
```

referring to frame 4660 (hexadecimal 1234 = decimal 4660). The preceding full stop indicates that the following address is to be specified as a hexadecimal number.

4.4 Frames and data

In order to understand the concept of addresses and displacement, let us look at the way in which data is held on disk.

If we were to inspect the contents of a typical frame of virtual memory storage, we might find that it looks like

```
FID: 10136 : 0 10502 47771 0 ( 2798 : 0 2906 BA9B 0 )

  1 :OAK^15^LS/15/77^5000^25^2000^8340^_00602200^SETTEE:
 51 :, BROWN, OAK^36]20]10^DN/1/43]LS/4/65]DN/11/70^100:
101 :0^30^2000]1000^8320]8321]8312^_00384444^SETTEE, BL:
151 :ACK, ASH^^DN/6/81^1000^30^1000^8320^_00511200^DESK:
201 :, GREY, ASH^16]32^LS/17/1]LS/17/2^5600^30^1000]300:
251 :0]2000^8320]8302^_003C4500^CHAIR, RED, LAMINATE^18:
301 :^LN/3/29^2500^60^1000^8345^_00409000^SIDEBOARD, YE:
351 :LLOW, ASH^99^DN/14/70^13000^15^1000^8345^_00415656:
401 :^SETTEE, BLUE-GREEN, MAPLE^6^DN/56/40^1000^30^2000:
451 :^8306^_00361234^CHAIR, AUBERGINE, MAHOGANY^7^DN/5/:
```

This display was produced on a PC implementation of Pick R83 version 3.1, and shows that the frames here are 512 bytes in length (12 bytes for the control information which we discuss in a moment plus 500 bytes for the data). On some systems, the frames are 1024 bytes long (24 control bytes plus 1000 data bytes), on others they are 2048 bytes long (40 control bytes plus 2000 data bytes).

By inspection, this frame would seem to be somewhere in the *middle* of the group since the data starts with what looks like the end of one record, and it ends with what appears to be the first part of another record. The numbers 1 to 451 show the position of the first character on that line within the data area of the frame. Thus, the letter O of OAK is in byte 1 of the data area, the character ^ (immediately before the 8306) is in byte 451 of the data area, and the final / character is in byte 500 of the data area. Strictly speaking, we should add 12 to these figures to allow for the 12 control bytes at the front of the frame.

As this illustrates, each frame of disk space consists of

* the control or linkage bytes. These are first 12 bytes of each frame and enable that frame to be associated with its overflow frames.

  In this instance, the actual linkage bytes are not displayed, but their contents are shown on the first line:

  FID: 10136 : 0 10502 47771 0 (2798 : 0 2906 BA9B 0)

  This tells us that

  + this shows the contents of frame 10136 (or frame 2798 in the hexadecimal notation);

  + that the data in this frame is continued from that in frame 47771 (BA9B). This information is known as the **backward link**;

  + that this data in this frame overflows into frame 10502 (2906). This information is known as the **forward link**.

  These are the first 12 bytes (of a 512-byte frame), the first 24 bytes (of a 1024-byte frame), and so on.

* the data items.

  The general format of a physical item is:

      cccciii^ddd^eee^fff^ggg^_

  where **cccc** is the four-byte item length count, **iii** is the item-id, **ddd, eee, fff** and so on are the data attributes of the item and the character shown here as _ is the end-of-item indicator.

In this illustration, familiarity with the data contents allows us to identify that the section

```
| 101 :0^30^2000]1000^8320]8321]8312^_00384444^SETTEE, BL: |
| 151 :ACK, ASH^^DN/6/81^1000^30^1000^8320^_00511200^DESK: |
```

contains the physical item

00384444^SETTEE, BLACK, ASH^^DN/6/81^1000^30^1000^8320^_

The item length counter (0038 in this instance) is maintained by the operating system whenever the item is written to disk and tells the operating system the exact physical length of this item. The item length counter is held as a hexadecimal number (0038 hexadecimal is the equivalent of decimal 56, in this instance). This field gives the total length of

+ all the data attributes and field separators, plus

+ the item-id, plus

+ the end-of-item marker, plus

+ the four-byte count field itself.

The data in this particular example indicates that

+ the item-id is 4444

+ attribute 1 contains SETTEE, BLACK, ASH
+ attribute 2 is null
+ attribute 3 contains DN/6/81
+ attribute 4 contains 1000
+ attribute 5 contains 30
+ attribute 6 contains 1000
+ attribute 7 contains 8320

* the end-of-data marker. This is a final marker indicating that there is no further data belonging to this group.

  This is one of the system separator characters. On most implementations, as in the illustration shown below, this is the segment mark (character 255), on other implementations, it may be the attribute mark (character 254).

If we were to look at the *last* frame of the group, like this:

```
      FID: 23981 : 0 0 18131 0 ( 5DAD : 0 0 46D3 0 )

  1 :^_00584200^SETTEE, GOLD, ASH^55]10^XN/19/35]LS/6/6:
 51 :^1000^30^1000]2000]3000]4000^8343]8345^_004C8117^S:
101 :IDEBOARD, GREY, OAK^0]0^MN/5/66]LN/91/55^13000^15^:
151 :1000^8300]8339^_005B2936^SETTEE, GREY, OAK^36]20]1:
201 :0^DN/1/2]LS/4/1]DN/11/1^700^30^2000]1000^8344]8316:
251 :]8338^_003D1000^DESK, GREEN-BLUE, ASH^8^MN/17/81^5:
301 :600^30^2000^8346^__..............................:
351 :..................................................:
401 :..................................................:
451 :..................................................:
```

we would observe several points:

* this is frame 23981 and it is a continuation of the data in frame 18131 (as shown by the backward link), but the data is not continued into any further frames (the forward link is zero).

* the final item in the file is 003D (hexadecimal bytes) in length with the item id 1000, and the data contents are

  DESK, GREEN-BLUE, ASH^8^MN/17/81^5 600^30^2000^8346^

* of the two final _ characters, the first is the end-of-item marker, and the second is the end-of-group marker.

* the rest of the frame (shown here as dots) is of no concern to our file and is just what happened to be in this frame when it was last used prior to being seized for use as overflow for our file.

```
FID:  12345 :  0  0  0  0          ( 3039 : 0  0  0 0 )

  1 :003D1000^DESK, GREEN-BLUE, ASH^8^LS/17/81^5600^30^:
 51 :2000^8205^_003C2000^SETTEE, YELLOW, OAK^18^DN/1/69:
101 :^10000^30^1000^8176^_003D3000^SIDEBOARD, BLUE, ASH:
151 :^58^MN/5/56^13000^15^2000^8178^_003A4000^DESK, BLA:
201 :CK, MAPLE^68^LS/7/87^5600^30^1000^8173^_003B5000^S:
251 :ETTEE, ORANGE, ASH^24^DN/19/3^1000^30^1000^8180^__:
301 :..................................................:
351 :..................................................:
401 :..................................................:
451 :..................................................:
```

The locations of various pieces of data in this particular instance are as follows:

* the four-byte count field 003D of the first item starts in the very first data byte at displacement 12 (that is, the displacement is 1+12-1 = 12).

We use the number 12 because the first twelve bytes of each frame are used for the control and linkage information, to show how this frame is associated with any other frames in that group of the file. The actual frame comprises 512 bytes, these 12 control bytes and then 500 bytes for data. Those systems which use 1024-byte frame, 24 bytes are used for control purposes, leaving 1000 for data; some implementations use a 2048-byte frame with 48 bytes for control information and 48 bytes for data. In this situation, you would use 24 or 48, respectively, where we use 12 in our calculations here.

* the four-bytes count field 003C of the second item starts at displacement 73 (that is, 62+12-1)

* the end-of-data marker after the very last item starts at displacement 311 (that is, 300+12-1)

4.5 Displacement

Having specified the FID of the frame, the position within the frame of the data which we are interested in is specified by means of the **displacement**. Thus, the first byte in frame 4660 has a displacement of 0 and is identified by the address

    4660,0

When calculating the displacement, you should remember that the first 12 (or 24 or 48) bytes of the frame which are used to held the linkage information; these must be included in the displacement count, even if they are not shown on the display produced by the TCL DUMP command. Thus, if you wish to patch the first bytes of **data** in a 512-byte frame, this will have a displacement of 12 (not 0 and not 1):

    4660,12

If you wish to specify the displacement in *hexadecimal*, then you will use the full stop instead of the comma:

    4660.B

or the equivalent forms

    .1234.B

or

    .1234,12

Since you will frequently use the TCL DUMP command to locate the data which you want to inspect, you should be aware of the manner in which the DUMP output is displayed. It may be as shown above, where the control/linkage bytes are not shown and the numbering starts at 1. Some other displays formats are:

1) showing the linkage bytes and numbering the bytes from 0.

```
000 ............002AS100^MICHAEL SMITH^34 HIGH ST^ANYT
050 ON^_0027B100^MARY BROWN^12 MAIN RD^BEATON^__??????
100 ??????????????????????????????????????????????????
150 ??????????????????????????????????????????????????
200 ??????????????????????????????????????????????????
250 ??????????????????????????????????????????????????
300 ??????????????????????????????????????????????????
350 ??????????????????????????????????????????????????
400 ??????????????????????????????????????????????????
450 ??????????????????????????????????????????????????
500 ????????????
```

In this case, the displacement is derived directly from the display. Thus the O of ANYTON has a displacement of 50.

2) not showing the linkage bytes and numbering the data bytes from 0.

```
000 002AS100^MICHAEL SMITH^34 HIGH ST^ANYTON^_0027B100
050 ^MARY BROWN^12 MAIN RD^BEATON^__??????????????????
100 ??????????????????????????????????????????????????
150 ??????????????????????????????????????????????????
200 ??????????????????????????????????????????????????
250 ??????????????????????????????????????????????????
300 ??????????????????????????????????????????????????
350 ??????????????????????????????????????????????????
400 ??????????????????????????????????????????????????
450 ??????????????????????????????????????????????????
```

In this case, the displacement is derived by adding 11 (that is, adding 12 and subtracting 1) to the number shown on the display. Thus the ^ immediately before MARY BROWN has a displacement of 61 (50 plus 11).

3) showing the linkage bytes and numbering the bytes from 1.

```
001 ............0029S100^MICHAEL SMITH^34 HIGH ST^ANYT
051 ON^0026B100^MARY BROWN^12 MAIN RD^BEATON^^????????
101 ??????????????????????????????????????????????????
151 ??????????????????????????????????????????????????
201 ??????????????????????????????????????????????????
251 ??????????????????????????????????????????????????
301 ??????????????????????????????????????????????????
351 ??????????????????????????????????????????????????
401 ??????????????????????????????????????????????????
451 ??????????????????????????????????????????????????
501 ????????????
```

In this case, the displacement is derived by subtracting 1 from the number shown on the display. Thus the O of ANYTON has a displacement of 50 (51 minus 1), and the ^ immediately before MARY BROWN has a displacement of 61 (62 minus 1).

If the frame-size on your implementation is larger, or the display numbering differs, then you must amend your calculations accordingly.

4.6 Window

A third piece of information which may be required is the window. This is simply the length of the data which is to be displayed or changed.

This can be expressed as a decimal number:

```
10
```

or as a hexadecimal number

```
.A
```

4.7 Address and window - examples

Here are some further examples of FID, displacement and window specifications:

```
75169,0;12
     to display the 12 bytes start starting at the first
     byte of frame 75169

61986,374;.58
     to display the 88 bytes start starting at the 375th
     byte of frame 61986

33397.1AE;73
     to display the 73 bytes start starting at the 431st
     byte of frame 33397

67918.C2;.11A
     to display the 282 bytes start starting at the 195th
     byte of frame 67918

.8447,339;82
     to display the 82 bytes start starting at the 340th
     byte of frame 33863

.3E12,40;.5D
     to display the 93 bytes start starting at the 41st byte
     of frame 15890

.E5BE.8;500
     to display the 500 bytes start starting at the 9th byte
     of frame 58814

.10E5F.158;.2B
     to display the 43 bytes start starting at the 345th
     byte of frame 69215
```

5 Inspecting frames of virtual memory

The C, I and X commands of the interactive system debugger are used to display and/or change the contents of any part of the disk storage.

These commands will include the address (the location) of the first character of the data to be displayed and the window (the length) of the string of data to be displayed.

Caddress;window
: will display the contents of the virtual memory (the number of bytes is specified by **window**) starting at position specified by **address**, in **character** format, and allow the user to amend the data.

  For example, the command

  ```
  C1234,56;10
  ```

  will display the 10 bytes starting at byte 56 of frame 1234, outputting the data in character format. Not graphic characters will be shown as full stops.

Xaddress;window
: will display the contents of the virtual memory (the number of bytes is specified by **window**) starting at position specified by **address**, in **hexadecimal** format, and allow the user to amend the data.

  For example, the command

  ```
  X1234,56;10
  ```

  will display the 10 bytes starting at byte 56 of frame 1234, outputting the data in hexadecimal format.

The format of the address and the window are discussed above. If the window specification is omitted, then the size of the last window specification will be assumed.

In each case, the debugger will display the current contents of the specified window of the frame in the required format, and then invite the user to enter the new contents of that window. For example, if we issue any of the above commands, the sequence might look like this:

```
!C12345,56;10 44-7FF^9C0=

!X12345,56;10 .34342D374646FE394330=
```

showing the contents of the 10 bytes in character format and hexadecimal format, respectively.

At this point, there are several possible courses of action. You may abandon the process, look at a further window or change the data in the window. Your choice is one of the following commands:

<RETURN>
: will leave the data unchanged and return to the !_ prompt.

<LINE FEED>
will leave the data unchanged and pass on to display the data in the next window.

<CTRL>N
will leave the data unchanged and then pass on to the next window, displaying the address and the data there. The letters EOF indicate that the end of the frame has been reached.

<CTRL>P
will leave the data unchanged and then pass on to the previous window, displaying the address and the data there. The letters EOF indicate that the start of the frame has been reached.

C<LINE FEED>
C<RETURN>
will redisplay the window in character format.

I<LINE FEED>
I<RETURN>
will redisplay the last two bytes of the window as a decimal number.

X<LINE FEED>
X<RETURN>
will redisplay the window in hexadecimal format.

In the following section, we see how to change the data in the window.

5.1 Changing frames of virtual memory

When the Caddress;window or Xaddress;window command has been issued and the appropriate data displayed, the debugger will give the user an opportunity to change that data:

```
C1234,56;7 12..34A=
```

showing that the seven bytes starting at the 57th byte of frame 1234 currently contain the characters

```
12..34A
```

If the contents of the window are to be changed,

AND THIS SHOULD ONLY BE DONE BY AN EXPERIENCED PICK USER AND WITH FULL AWARENESS OF THE POTENTIALLY DISASTROUS CONSEQUENCES OF MAKING A MISTAKE,

then the replacement data may be now be entered. The replacement data may be expressed in any of the following forms:

'ccccc<RETURN>

will replace the data in the window by the characters ccccc. The apostrophe indicates that **character** data follows.

.xxxxxxxx<RETURN>
will replace the data in the window by the hexadecimal string **xxxxxxxx** - there must be an even number of hexadecimal digits. The full stop indicates that **hexadecimal** data follows.

Less likely, are the these responses:

0<RETURN>
will replace the data in the window by a string of null characters (hexadecimal 00).

The fact that only the number 0 need be entered, thereby setting the entire contents of the window to nulls, makes this a most dangerous feature of the system debugger, and one that should only be used by technical users who really know what they are doing.

n<RETURN>
will replace the data in the window by the integer value **n**. Since it is most unlikely that anyone (except an Assembly language programmer) would know the integer equivalent of any particular string, this form is rarely used.

If the replacement data consists entirely of the normal keyboard characters, then this may be conveniently expressed as characters by means of the

'cccccccccc

specification.

If the correct data, which is to be entered, contains any special characters, such as

field-separators,
attribute marks,
value marks,
subvalue marks,
segment marks,
the end-of-item marker, or the
end-of-data marker,

then the replacement data is most conveniently expressed in hexadecimal notation by means of the

.xxxxxxxxxx

specification. The hexadecimal equivalents of the various characters can be found in a table of ASCII characters.

In all cases (except for the single 0 specification), the new entire string of data is used irrespective of the length of the string specified by the window. Thus, if the new string is *shorter* than the original string which was specified by the window, then the additional characters will be unchanged, and if the new string is *longer* than the original string, then the extra characters outside the window will be replaced.

Below, we discuss some applications of this technique in patching a frame.

5.2 Patching a frame

In the *MB-Guide to Group Format Errors*, we see that a final - and most desperate - means of recovering data which has been lost as a result of a **group format error** is to use the interactive system debugger to **patch** the offending frame, replacing the actual incorrect bytes by correct data.

We include this technique here to illustrate an application of the interactive system debugger.

In order to patch a frame you must know:

* the exact location of the **incorrect** data. This is the **address** of the data.

* the length of the incorrect data. This is the **window**.

* the contents of the incorrect data. This enables you to confirm that you have displayed (and will be changing) the string which you want to change.

* the exact contents of the **correct** data. You should write down the exact characters which you want to put into the frame.

Armed with this information, you are now ready to patch the frame. The steps to be taken are as follows:

1) use the TCL DUMP command to print a copy of the data contents of the frame before you start to make any changes. For example, the command

    DUMP 12345 XP

   will print the contents of frame 12345 in character and also in hexadecimal format. The hexadecimal format will show any characters will cannot be printed.

2) enter the interactive system debugger by hitting the <BREAK> key.

   You can do this from any terminal and at any point in your processing: in the middle of a process or, preferably, at TCL.

3) when you get the ! prompt character, enter your specifications in the form

   f,d;w

where **f** is the FID of the frame which is to be patched; **d**is the displacement of the first byte in the string which is to be changed; **w** is the window (the length of the data string to be changed).

Pay particular to the punctuation here: a **comma** follows the FID and a **semi-colon** follows the displacement. An invalid command will be rejected by the interactive system debugger.

4) the debugger will then display the **w** characters starting at byte displacement d. Confirm that this is the offending string.

   If this is the wrong string, hit <RETURN> and try again from step (3).

5) Enter the correct data as a series of **pairs** of hexadecimal characters (preceded by a full stop) or as a series of display characters (preceded by an apostrophe). For example:

   .FF

   or

   '003D

6) When you are satisfied that you have entered the correct data, hit the <RETURN> key.

7) Abandon the interactive system debugger by entering

   END

   followed by the <RETURN> key.

8) DUMP a copy of the frame after you have made the changes and confirm that the action has taken place successfully.

5.3 Patching a frame - some examples

In this section, we present some examples of the use which the experienced technical user is likely to make of the interactive system debugger.

We would again stress the care which should be taken when patching any frames.

5.4 Example 1: clearing a frame

Let us suppose that we wish to set an end-of-data marker in the first data byte of this frame:

```
FID:  12345 :  0  0  0  0          ( 3039 : 0  0  0 0 )

  1 :003D1000^DESK, GREEN-BLUE, ASH^8^LS/17/81^5600^30^:
 51 :2000^8205^_003C2000^SETTEE, YELLOW, OAK^18^DN/1/69:
101 :^10000^30^1000^8176^_003D3000^SIDEBOARD, BLUE, ASH:
151 :^58^MN/5/56^13000^15^2000^8178^_003A4000^DESK, BLA:
201 :CK, MAPLE^68^LS/7/87^5600^30^1000^8173^_003B5000^S:
251 :ETTEE, ORANGE, ASH^24^DN/19/3^1000^30^1000^8180^__:
301 :..................................................:
351 :..................................................:
401 :..................................................:
451 :..................................................:
```

This will effectively cancel this and the remaining frames in the group. The first part of the dialogue with the debugger might look like this:

```
!C12345,12;1 0=
```

the interactive system debugger indicating that this byte currently contains a zero. To change this to an end-of-data marker (hexadecimal FF in this instance), I should enter the

```
.
```

to indicate that I am specifying the data as a set of hexadecimal digits, followed by

```
FF
```

(the hexadecimal equivalent of the end-of-data marker), and finally I should hit the <RETURN> key. The display might look like

```
!C12345,12;1 0=.FF<RETURN>
```

If you feel that you will be likely to need to practise this technique, then it is a good plan to create a dummy file and a large dummy item with recognisable data and use this to patch parts of the **data** of the item. This will be perfectly safe, provided that you do not alter the item-length counter or the final end-of-item marker. When you have practised sufficiently, you should delete the file.

5.5 Example 2: clearing the backward/forward links

Let us suppose that some problem has occurred with the result that the forward and backward link of this frame have become corrupted:

```
FID 1234 : 0 -382533399 1023469895 0 (4D2 : 0 3300E9 E947 0 )

  1 :003D1000^DESK, GREEN-BLUE, ASH^8^LS/17/81^5600^30^:
 51 :2000^8205^_003C2000^SETTEE, YELLOW, OAK^18^DN/1/69:
101 :^10000^30^1000^8176^_003D3000^SIDEBOARD, BLUE, ASH:
151 :^58^MN/5/56^13000^15^2000^8178^_003A4000^DESK, BLA:
201 :CK, MAPLE^68^LS/7/87^5600^30^1000^8173^_003B5000^S:
251 :ETTEE, ORANGE, ASH^24^DN/19/3^1000^30^1000^8180^__:
301 :..................................................:
351 :..................................................:
401 :..................................................:
451 :..................................................:
```

To overcome the problem, we must reset the entire control/linkage bytes to 0. The 12 control/linkage bytes start at displacement 0, so we could do this by means of the sequence

```
X1234,0;12
```

We use the X command since the C would show the data in character format and, in this instance, most of these would be non-printable characters. The debugger's response would be

```
.2600E93300E93D00E94700E9=
```

and we could set this entire string to null by entering

```
0<RETURN>
```

So the entire sequence would look like this on the screen:

```
!X1234,0;12 .2600E93300E93D00E94700E9=0
```

5.6 Example 3: recovering a lost item

As our third example, let us suppose that we have inadvertently deleted item 5000 of a file. But, since the item lay at the end of its group, it is still visible when we dump the appropriate frame:

```
FID:  12345 :  0  0  0  0          ( 3039 : 0  0  0 0 )

  1 :003D1000^DESK, GREEN-BLUE, ASH^8^LS/17/81^5600^30^:
 51 :2000^8205^_003C2000^SETTEE, YELLOW, OAK^18^DN/1/69:
101 :^10000^30^1000^8176^_003D3000^SIDEBOARD, BLUE, ASH:
151 :^58^MN/5/56^13000^15^2000^8178^_003A4000^DESK, BLA:
201 :CK, MAPLE^68^LS/7/87^5600^30^1000^8173^__03B5000^S:
251 :ETTEE, ORANGE, ASH^24^DN/19/3^1000^30^1000^8180^__:
301 :..................................................:
351 :..................................................:
401 :..................................................:
451 :..................................................:
```

It is only the end-of-group marker - that is the second _ character in this sequence:

```
| 201 :CK, MAPLE^68^LS/7/87^5600^30^1000^8173^__03B5000^S: |
```

which prevents us from accessing the following item, 5000. By inspection, we see that the four-byte item-length counter for item 5000 should be

```
003B
```

so by replacing the end-of-group marker by a 0, we can re-instate item 5000.

The offending _ character is in position 241 on the DUMP display. Adding 12 (the length of control/linkage field) and subtracting 1 (the number of the first data byte), we see that this has a displacement of 252. So our debug command would be:

```
C12345,252;1
```

The debugger's response would be

```
_=
```

and we could replace this by 0 by entering

```
'0<RETURN>
```

So the entire sequence would look like this on the screen:

```
!C12345,252;1 _='0
```

The item will now be accessible once again. Obviously this method of recovering a lost item can only be used when

* the item was the last item in its group before it was deleted (otherwise, all the trailing items would have been shifted up and overwritten the item when it was deleted).

* the item did not overflow into another frame (otherwise, the overflow frames would have been released when the item was deleted).

6 Arithmetic and conversion commands

There are several TCL commands which can be issued from within the interactive system debugger. These are provided as a tool and allow to convert decimal/hexadecimal values without leaving the debugger.

ADDD decimalnumber1 decimalnumber2
to add together two decimal numbers and display the result.

ADDX hexnumber1 hexnumber2
to add together two hexadecimal numbers and display the

result.

DIVD decimalnumber1 decimalnumber2
to divide one decimal number by another and display two results: the quotient and the remainder.

DIVX hexnumber1 hexnumber2
to divide one hexadecimal number by another and display two results: the quotient and the remainder.

DTX decimalnumber
to convert a decimal number to hexadecimal and display the result.

DTX base decimalnumber
to convert a decimal to any other base and display the result. If the base is omitted, 16 is assumed.

MULD decimalnumber1 decimalnumber2
to multiply together two decimal numbers and display the result.

MULX hexnumber1 hexnumber2
to multiply together two hexadecimal numbers and display the result.

SUBD decimalnumber1 decimalnumber2
to subtract one decimal number from another and display the difference.

SUBX hexnumber1 hexnumber2
to subtract one hexadecimal number from another and display the difference.

XTD hexnumber
to convert a hexadecimal number to decimal and display the result.

XTD base basenumber
to convert a number from any other base to decimal and display the result. If the base is omitted, 16 is assumed.

The following table illustrates some examples of these commands:

Command	Result
ADDD 987 432	1419
ADDX FACE CAFE	1C5CC
DIVD 987 432	2 123
DIVX FACE CAFE	1 2FD0
DTX 98765	181CD
DTX 32 98765	30ED
MULD 987 432	426384
MULX FACE CAFE	C6DF6464
SUBD 987 432	555
SUBX FACE CAFE	2FD0
XTD FACE	64206
XTD 32 FACE	502158

7 The debug commands

The primary purpose of the interactive system debugger is for the identification, location and correction of errors in Assembly language routines. In this respect, the interactive system debugger offers similar options to the Basic symbolic debugger:

* allowing the processing to interrupt when a specified number of instructions have been executed;
* allowing the processing to interrupt when any one of up to four breakpoints is reached.
* allowing the programmer to display specific data elements and registers.
* allowing the programmer to execute the process step by step, and to specify the size of the steps.
* allowing the programmer to display the table of breakpoints and traces.

This area is of interest to Assembler language programmers and is therefore outside the scope of this present beginner's guide, but we summarise the available commands here:

B {address}
: will set a breakpoint at the address specified.

The K command will kill a breakpoint which has previously been set by the B command.

B fid.0
: will set a breakpoint at every entry point in the frame specified.

D
: will display the current table of breakpoints and traces.

DB

This is only available from the SYSPROG account and switches the interactive system debugger facilities on and off. When the interactive system debugger is off, the facilities to display and/or change parts of virtual memory cannot be used.

E {instructions}
will set the number of instructions which are to be executed between interrupts.

E
will cancel any previous Einstructions setting.

G {address}
will cause the processing to continue from the **address** specified. **Address** is specified in the format shown below. If **address** is omitted, the processing will continue from the point of interruption.

K {address}
will kill a breakpoint which has previously been set by the B command.

L {address}
will display the linkage information for the frame specified by the address.

If the address is omitted, the linkage information for the current frame - at which the interrupt occurred - will be displayed.

M {address}
will switch the modal trace on/off. When the modal trace is active, the processing is interrupted each time a new frame is entered.

ME {port}
will assign all

N {count}
will continue processing until the specified number of breakpoints have been passed.

T {address}
will add an entry to the trace table so that the contents of the data element and its address will be displayed each time the element changes.

U {address}
will cancel the trace table entry added by the most recent T command. If the address is omitted, then all entries will be cancelled.

Y {address}
will add an entry to the Y-trace table so that the contents of the data element and its address will be displayed each time the element changes.

Z {address}

will cancel the Y-trace table entry added by the most recent Y command. If the address is omitted, then all entries will be cancelled.

8 Error messages

All debugger commands must be entered in upper-case, as indicated here.

The interactive system debugger error messages are fairly laconic.

BAD CHAR
: is displayed when the response to a C or X command is not of the correct format.

CMND?
: is displayed in many situations:

* if the debugger is not available.
* if an invalid command is entered,
* if a command with the wrong format is entered.

ILLGL SYM
: if an invalid command is entered.

WINDOW
: if an invalid window is specified, or if an invalid response is given when displaying a window.

Glossary

The following terms are used in this *MB-Guide*:

In this and the other *MB-Guides*, the keyboard control keys have been represented by their name enclosed in angle brackets:

<BREAK> identifies the break key or the equivalent sequence which interrupts the current process.

<CTRL> identifies the control key.

Certain characters are entered at the keyboard as a combination of one or more of the above keys together with other keyboard characters. For example, the subvalue-mark (character 252) may be entered as:

<CTRL> \

that is, by holding down the <CTRL> key and typing the normal \ character at the same time. Similarly, the value-mark can be keyed in as the sequence <CTRL>] and the attribute-mark as the sequence <CTRL> ^

<ENTER> identifies the ENTER key which is used to transmit each piece of data to the system.

This is generally represented by the <RETURN> key in the text.

<ESC> identifies the ESCAPE key.

<LINE FEED> identifies the line-feed key. On some keyboards, this may be the down-pointing arrow. The sequence <CTRL> J is equivalent.

<RETURN> identifies the RETURN key which is used to transmit each piece of data to the system.

On some keyboards, this may be the <ENTER> key or the down-left-pointing arrow key. The sequence <CTRL> M is equivalent.

Index

MB-Guides

MB-Guides are designed to serve as introductory texts to a range of fundamental topics within the Pick operating system. They will be available for the following subjects:

MB-Guide to Access conversions and correlatives
MB-Guide to Access sentences
MB-Guide to Basic programming
MB-Guide to Creating and using Procs
MB-Guide to using the Editors
MB-Guide to File design
MB-Guide to File-save and file-restore
MB-Guide to Files: monitoring and sizing
MB-Guide to Group format errors
MB-Guide to Operations and systems management
MB-Guide to Pick on the PC
MB-Guide to Program design
MB-Guide to Security
MB-Guide to The Basic symbolic debugger
MB-Guide to The spooler
MB-Guide to The system debugger
MB-Guide to Using backing storage

The format of the *MB-Guides* is such that they may be easily updated and amended to reflect the current state of the operating system. In order that this and the other Mating MB-Guides continue to meet the needs of the users, we would appreciate your comments on this guide and your suggestions for further titles in this series.

MB-Master self tuition courses are also available on a wide range of topics related to the Pick operating system:

Access techniques
Advancing in Basic
Moving to Basic - a conversion course
Pick systems management
Programming in Basic
Starting Access
Starting ACCU/PLOT
Starting CompuSheet+
Starting Jet
Starting Pick
Starting Runoff
Starting SB+
Systems development
Writing Procs

If you have any comments on this *MB-Guide* or any suggestions for further title in the series, then please send your suggestions to:

MALCOLM BULL
MALCOLM BULL TRAINING AND CONSULTANCY SERVICES
19 Smith House Lane
BRIGHOUSE
West Yorkshire
HD6 2JY

Telephone: 0484-713577 **Fax: 0484-714112**

We are particularly concerned about:

* any additional information which you would like.
* any topics which you feel are superfluous.
* any inaccuracies which you noticed.
* any areas in which the information given in the *MB-Guide* did not apply to your implementation. Please indicate what implementation you were using.
* any other suggestions, observations and comments.

If you do write to us, please give your name and address so that we can acknowledge your contribution in the next Edition of this *MB-Guide*.

MB-Guides

The booklets in the *MB-Guide* series cover a range of fundamental topics of interest to users and those responsible for running Pick systems.

Each *MB-Guide* deals with a specific aspect of the operating system and the booklets represent an economical introduction to the various topics and the whole series forms an integrated presentation of the subject matter.

The booklets are intended to be a working document and, for this reason, space is provided for the user's notes, and the reader is encouraged to amend the booklet so that it applies to his/her own system.

It is anticipated that the series of *MB-Guides* will be of special interest to new users, and it should prove useful for software houses and others who are responsible for the instruction of their clients and staff in the fundamental aspects of the Pick operating system.

MB

Malcolm Bull

Training and Consultancy Publications