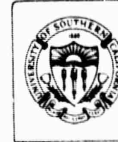
(6)

AD A048265



ISI/TM-77-5
October 1977

ARPA ORDER NO. 2223

PRIM System:

■ AN/UYK-20 User Guide
■ User Reference Manual

DDC
DEC 21 1977

Louis Gallenson
Alvin Cooperband
Joel Goldberg

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

AD NO.
DDC FILE COPY

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

Best
Available
Copy

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/TM-77-5	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PRIM System: AN/UYK-20 User Guide and User Reference Manual.		5. TYPE OF REPORT & PERIOD COVERED Technical manual, 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Louis Gallenson, Alvin Cooperband Joel Goldberg		8. CONTRACT OR GRANT NUMBER(s) DAHC 15-72-C-0308, ARPA Order-2223
9. PERFORMING ORGANIZATION NAME AND ADDRESS Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order #2223 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October 1977 13. NUMBER OF PAGES 53
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

debugging tool, emulated I/O, emulation-based programming tools, emulators, microprogramming

407 952

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

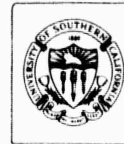
This is a two-part manual for users of the PRIM-based UYK-20 emulator. The manual demonstrates as well as describes the capabilities of PRIM, running and debugging of object code, and the emulated computer system.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



ISI/TM-77-5
October 1977

ARPA ORDER NO. 2223

PRIM System:

■ AN/UYK-20 User Guide
■ User Reference Manual

Louis Gallenson
Alvin Cooperband
Joel Goldberg

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223, PROGRAM CODE NO. 3D30 AND 3P10.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF ARPA, THE U.S. GOVERNMENT OR ANY OTHER PERSON OR AGENCY CONNECTED WITH THEM.

THIS DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE: DISTRIBUTION IS UNLIMITED.

Preceding Page Blank - Not Filmed

CONTENTS

AN/UYK-20 User Guide

Introduction *1*

Enter and debug a small program *3*

Bootstrap a program from paper tape *8*

Find which instructions modify a location *10*

Find which instruction sets a location to a value *11*

Determine how many times a code sequence is entered *11*

Count references prior to a designated condition *12*

Trace a loop only once *12*

Determine which instructions were NOT executed *13*

Determine when data change over a code sequence *13*

Search a buffer for a given value *14*

Appendix: UYK-20 Reference Listing *15*
- Debugger numbers and operators *15*
- UYK-20 parameters and devices *15*
- UYK-20 symbols and cells *16*
- Breakpoints *19*

NTIS
DDC
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

User Reference Manual

Introduction 1
General input conventions 1

PRIM Exec 3

PRIM Debugger 14
Arguments 14
Values 14
Expressions 14
Expression ranges 15
Lists of expressions or ranges 15
Spaces 15
Syntactic units 15
Literals 16
Symbols 16
Punctuation 16
Error detection and editing 17
Commands 17
Debugger control 17
Execution control 20
Display 22
Storage 24

Target Execution State 25
Target I/O 25
I/O error messages 26

PRIM SYSTEM: AN/UYK-20 USER GUIDE

INTRODUCTION

The PRIM system supports UYK-20 program development and testing by providing an emulated UYK-20 tool embedded in an interactive time-sharing environment. This emulated UYK-20 provides powerful debugging aids not possible on an actual UYK-20 computer system.

This guide consists of two sections, serving distinct purposes. This first section is an extended introduction to the PRIM UYK-20 tool and its capabilities; it is addressed to the UYK-20 user with no prior exposure to PRIM. It consists of an overview of the tool, followed by a detailed discussion of a number of common or representative programming problems with solutions illustrated by means of actual session transcripts with the PRIM UYK-20 emulation tool. The second section is an appendix to a separate document, ***The PRIM System: User Reference Manual***; that manual and the appendix together constitute the complete reference document for the PRIM UYK-20 tool. (The PRIM system supports a family of emulation tools; the ***User Reference Manual*** covers the capabilities of the PRIM system as they apply to all the tools in general.)

PRIM is available through both the NSW (National Software Works) and the USC-ISIC TENEX system, which is a server system on the ARPANET. The user of PRIM is assumed to have access to one or the other system and some rudimentary familiarity with its use. Once the user enters PRIM, its behavior is identical in the two environments.

PRIM consists of the emulated UYK-20 plus two separate command interpreters known as the exec and the debugger. At any time, PRIM is either running the UYK-20 emulator or processing user exec or debugger commands; the transition between states is at the control of the user.

Exec commands are concerned primarily with the manipulation of UYK-20 environments and configurations. The elements of exec commands are keywords, file names, and (decimal) numbers. Keywords include such items as command names, device names, options, and parameters. They need not be entered in their entirety; any unambiguous leading substring of the desired word suffices for recognition. (When a keyword is terminated with an ***escape*** character, the word is completed by the exec.) File names refer to files in the user's file space (in either NSW or TENEX), and follow the appropriate file name syntax. Each file specification requires the name, as appropriate, of either an existing file (to be read or modified) or a new file (to be created and written).

Debugger commands are concerned with the detailed control of the emulated UYK-20. The debugger includes the functions available on the front panel of the UYK-20 as a small subset of its capabilities. Debugger commands each consist of a single character; the arguments to those commands are symbolic expressions which name the elements of the UYK-20 (e.g., memory locations, registers, PC, channel control memory).

Within PRIM, certain ASCII characters have been assigned special functions when input by the user. These functions, which are described completely in the ***User Reference Manual***, concern command editing and PRIM (command and UYK-20) control. The command editing functions are backspace (either ***backspace*** or ***cntl-A***), backup (***cntl-W***), delete (***del***

editing functions are backspace (either *backspace* or *cntl-H*), backup (*cntl-W*), delete (*del* or *rubout*), retype (*cntl-R*), and question (*question-mark*); the control functions are status (*cntl-S*) and abort (*cntl-X*). Backspace backs up over one character within the current field of a command; it is acknowledged by a backslash (\) followed by the erased character. Backup backs up over the current command field; it is acknowledged by a backslash (\) followed by the first character of the erased field. Delete backs out of the current subcommand entirely (or out of the current command if not in a subcommand); it is acknowledged by "XXX", followed by a new prompt. Retype re-displays the current command or subcommand line. Question, when entered as the first character of a field, generates a summary of the input currently expected, followed by a retype of the line.

Status causes PRIM to respond with the current status of the emulated UYK-20. Abort causes any operation in progress to be cleanly terminated and returns control to the top level of PRIM (either exec or debugger, depending on which one last had control). The abort function is used both to abort a command that is partially entered or in process and to stop the running UYK-20.

With this background we can now illustrate how PRIM can be used. Two examples will be explored in detail; these examples will show how to:

1. key in a small program, run it, discover a bug, fix the bug, trace it to observe its operation, and save the results,

2. load a program into memory from a paper tape.

Several further examples will be presented with considerably less detail to illustrate a number of ways in which the interactive PRIM debugger can be used. These examples will show how to:

1. find which instructions are modifying a location and which ones are setting it to a designated value,

2. determine how many times a code sequence is executed and how many times a data location is referenced prior to the occurrence of a known condition,

3. trace a loop only once,

4. find which instructions in a program were *not* executed,

5. test if a data location has changed over a code sequence, and

6. search a buffer for a given value.

In the following examples, drawn from actual PRIM session transcripts, user input is *italicized* to distinguish it from machine output. Input control characters appear as their abbreviations superscripted (e.g., ^{esc}).

ENTER AND DEBUG A SMALL PROGRAM

To start, let us follow a complete step-by-step sequence of interactions with PRIM. We shall enter a small program, debug it, then save the resulting memory image on a disk file for later use.

Our session begins with a command which requests the PRIM UYK-20 tool. From TENEX, we begin at the exec, whose prompt character is "@".

```
e<PRIM>UYK20cr
AN/UYK-20 (20/04/77)
Latest NEWS is 28-JUNE-77
>
```

From NSW, we begin at the front end with the equivalent command.

```
NSW: uescSE (tool named): uyk20escs (confirm): cr

   you will now be talking to 1UYK20


AN/UYK-20 (20/04/77)
Latest NEWS is 28-JUNE-77
>
```

In either case, the UYK-20 tool is loaded, publishes its greeting message, and enters the PRIM exec, whose prompt character is ">". The greeting includes two dates. The first is the release date of the current version of the UYK-20. The second is the date of the most recent item of on-line news (these on-line items may be accessed via the exec NEWS command, which is not demonstrated here); news items cover both new releases of the UYK-20 and changes to PRIM commands. The remainder of the session -- until we leave the UYK-20 tool at the end of the session -- is the same for both systems.

We direct the exec to keep a transcript of this session on a file, and then switch to the PRIM debugger, whose prompt character is "#", for the bulk of this sample session.

```
>TRescANSCRIPT (to file) EXAMPLEcr
>DescEBUG
#
```

The exec creates and opens a new file named EXAMPLE and records the remainder of the PRIM session on that file for subsequent reference by the user. Note that the TRANSCRIPT command itself is not actually found in the file since transcription begins only after the command is completed; it has been edited into this example.

For the transcribed record, we enter several lines of comments, preceding each with the PRIM comment character *semicolon.*

```
#; Routine to search a sorted table cr
#; by successive halving then selecting the half thatcr
#; should contain the given number.cr
#; cr
#;   Accepts:cr
#;     reg 2      the address of the first element in the tablecr
#;     reg 3      the address of the last element in the tablecr
#;     reg 4      the number to find in the tablecr
#;     reg 15     the return addresscr
#; cr
#;   Returns:cr
#;     reg 1      the address of the element in the table matching cr
#;                the contents of reg 4 or 0 if not foundcr
#Mode Expanded-lines cr

#Type 01000esc
 01000:   00 = LR 1,3\
 01001:   00 = SUR 1,2\
 01002:   00 = LLRS 1,1\
 01003:   00 = AR 1,2\
 01004:   00 = CI 4,1\
 01005:   00 = JER 15\
 01006:   00 = LJGE $+3\
 01007:   00 = LR 2,1\
 01010:   00 = LJ $+2\
 01011:   00 = LR 3,1\
 01012:   00 = CR 2,3\
 01013:   00 = LJNE 01000\
 01014:   00 = SUR 1,1\
 01015:   00 = JR 15cr

#; A small program to call the routine:cr
#Type 01200esc
 01200:   00 = LK 2,01300\
 01202:   00 = LK 3,01307\
 01204:   00 = JLK 15,0100C\
 01206:   00 = JS 01200cr
#; The ordered list of numbers:cr
#Type 01300esc
 01300:   00 = 0,2,4,7,9,20,80,100cr
```

Prior to entering the subroutine, we instruct the debugger to produce its output on separate lines using the Mode Expanded-lines command. (In the other choice, known as Dense-lines, the debugger compacts several lines of output into one.)

We begin building our subroutine at location 01000 (leading zeroes indicate octal numbers) by entering a type command with the desired address and terminating with an *escape.* In response, the debugger displays the contents of location 01000 followed by an equal sign to allow replacement. At this point we enter the first instruction of the subroutine followed by *backslash.* The debugger assembles and deposits the instruction at location 01000. It then displays the next available location, 01001, for replacement. In

this manner, successive instructions are deposited into successive locations. We type a *return* after the last instruction to terminate the type command.

Following the subroutine, we enter a small main program and a data table for testing the subroutine. The list of numbers, "0,2,4,...", each go into separate consecutive cells.

To begin the test, we load register 4 with a number the subroutine is to search for -- in this case 80 - then begin UYK-20 execution at location 01200.

```
#Set R4esc ← 80cr
#Go (to) 01200cr
--> Halted from 01206: JS 01200
- > AN/UYK-20 halted at 01200, Used 0:00.0
#Type R1:R3cr
R1: 00
R2: 01300
R3: 01300
#Type @R3cr
01300: 00
```

Within a short time, the emulator halts after executing the JS instruction at location 01206 and control returns to the debugger. We inspect registers R1 through R3. Register 1 contains a 0 indicating the subroutine's failure to find the value 80 in the table. Closer inspection reveals that, while register 3 should point to a value in the table greater than 80, it does not. The "@" is a unary operator meaning contents-of; the last type command above uses the @ operator to inspect the location addressed by R3 without having to enter the actual address.

Armed with this information, we place a breakpoint at the LJGE at 01006 to gain a closer view. We then restart the main program.

```
#Break (at) 01006esc (after doing) Xecute cr
#Go (to) cr
--> Break after executing 01006. LJGE 01011
#lf
--> Step from 01007: LR 02,01
#Type @R2,@R3cr
01300: 00
01303: 07
```

The break command instructs the UYK-20 to break -- suspend execution and return control to the debugger -- after every subsequent execution of an instruction at 01006. So, when the emulator finishes executing the conditional branch (whether it branches or not) control passes back to the debugger. At this point in the subroutine the location addressed by the contents of register 1 should have been compared and found to be less than 80 (register 4). Consequently, the subroutine should now search the interval between the address in R2 and that in R1. In other words, the address in R1 should replace R3 and the search repeated. We single-step the UYK-20 using the *line-feed* command and find that as the routine has been written, R1's contents replace R2 which, in effect, chooses to search the other half of the interval.

The solution is clear; we replace the instruction LJGE at 01006 with an LJLS instruction. We reset register 2 to the correct value and restart the subroutine at the comparison.

```
#Mode Instruction esc #Type 01006esc
 01006:   LJGE 01011 = LJLS 01011cr
#Type R2esc
 R2:   01300 = @R3\
 R3:   01303 = 01307cr
#Go (to) cr
- > Break after executing 01006:   LJLS 01011
#lf
--> Step from 01007:   LR 02,01
#Type @R2,@R3cr
 01305:   024
 01307:   0144
#Debreak (from) 01006cr
#Go (to) cr
--> Halted from 01206:   JS 01200
--> AN/UYK-20 halted at 01200, Used 0:00.0
#Type @R1cr
 01306:   0120
```

Again the breakpoint is reached and the program suspended. We single-step another instruction and observe the expected behavior of our subroutine. The breakpoint is removed and the routine is continued, eventually halting at the end of the main program. Inspection of register 1 reveals that the routine was successful in finding the desired value.

For the second test, we prepare the subroutine to search the table for a value known not to exist. After execution, several seconds elapse without any response from the program. We request and receive the status of our program by typing ↑S (which is not echoed). Suspecting the program to be looping infinitely, we type ↑X to suspend its execution.

```
#Set R4esc = 1cr
#Go (to) cr
--> AN/UYK-20 running at 01010, Used 0:00.9
↑X
- > AN/UYX-20 interrupted from running at 01000, Used 0:01.0
#Type R1:R3cr
 R1:   01301
 R2:   01300
 R3:   01301
```

Our inspection reveals that registers 2 and 3 correctly point to values in the table less than and greater than 1; the state of this program seems correct. So we must dig a bit further.

We build a break program to print registers 1 through 3 after the execution of each UYK-20 instruction. This allows us to observe the changes in the registers after each instruction execution.

```
#Break (at) .STEPesc
 ##Mode Dense-lines esc ###Type R1:R3cr
 ##cr
<Program number is [1]>   #Go (to) cr
--> R1: 01     R2: 01300    R3: 01301
--> Step from 01001:   SUR 01,02
#Go (to) cr
--> R1: 00     R2: 01300    R3: 01301
--> Step from 01002:   LLRS 01,01
#Go (to) cr
--> R1: 01300    R2: 01300    R3: 01301
--> Step from 01003:   RK 01,02
#Go (to) cr
--> R1: 01300    R2: 01300    R3: 01301
--> Step from 01004:   CI 04,01
```

The name .STEP is associated with the single-step event, instructing the UYK-20 to break after every instruction. The *escape* following .STEP indicates that we wish to associate a break-time program with the occurrence of this breakpoint. In response to the "##" prompt, we enter successive debugger commands which are not executed now, but rather saved by the debugger to be executed when the breakpoint occurs; the break program is terminated by an empty input line. Each subsequent Go then executes exactly one more instruction, followed by the break program.

After executing a few more instructions, we are able to determine the problem. The addresses in register 2 and 3 differ by 1. When the routine computes the address halfway between them, truncation occurs resulting in an address identical to that in register 2. The value in the table at this address is compared with 1, found to be less causing its address to be placed in register 2. Since the old address in register 2 is identical to the new one, no progress is made; the comparison for equality between the contents of registers 2 and 3 fail and the routine jumps back to 01000 to continue -- causing the infinite loop.

To correct this condition, we replace the section of code in the subroutine which updates register 2 or 3 to decrease the distance between the values by one. We restart the program.

```
#Mode Instruction cr
#Type 01006esc
 01006:  LJLS 01011 = LJLS $+4\
 01007:  LR 02,01 = LK 2,1,1\
 01011:  LR 03,01 = LJ $+3\
 01012:  CR 02,03 = LK 3,-1,1\
 01014:  SUR 01,01 = CR 3,2\
 01015:  JR 017 = LJLS 01000\
 01016:  00 = SUR 1,1\
 01017:  00 = JR 15cr
#Debreak (from) esc all [confirm]cr
#Go (to) 01200cr
```

When the program again terminates, we examine register 1 and find that the routine has correctly failed to find a nonexistent value in the table.

Satisfied with our program, we save the core image on the file BINSRCH.MEM and terminate our PRIM session.

```
--> Halted from 01206:   JS 01200
--> AN/UYK-20 halted at 01200, Used 0:01.0
#Type R1cr
 R1:   00
#Return (to EXEC) cr
>SAescVE ? One of the following:
 ALL
 CONFIGURATION
 MEMORY
 SYMBOLS
>SAVE MescEMORY (on file) BINSRCH.MEM;1cr
>QescUIT
    Quitting AN/UYK-20  [Confirm] cr
e
```

The file BINSRCH.MEM now contains a UYK-20 memory image that can be retrieved for subsequent use with the exec RESTORE command.

BOOTSTRAP A PROGRAM FROM PAPER TAPE

This example will demonstrate the process of loading a program from paper tape using the bootstrap loader.

The bootstrap program we need is the STANDARD NDRO, which is one of the NDRO's available in PRIM. This bootstrap assumes the existence of the 1532 input/output console on channel 1. Since the initial machine configuration contains no I/O devices, we must somehow install the console.

There are two methods available for attaching devices to the emulated UYK-20. The first is to retrieve the device assigments from a previously saved PRIM session using the RESTORE command. The second, which we will use, specifies each device individually. In either case, the installation of I/O devices must be done before the execution of any UYK-20 program in that session.

```
>SETcr
>>NDROesc ? STANDARD or SSIXS-A
>>NDRO STescANDARD cr
>>cr
>INSTALLesc (device) 1532esc (CHANNEL) 1cr
For the READER-1532,
>>SPEEescD (characters per second) 1000cr
>>cr
For the PUNCH-1532,
>>cr
For the TTY-1532,
>>cr
```

We set NDRO to STANDARD and install a 1532 on channel 1. During the installation of the 1532, we are prompted for parameters for each device connected to the controller. These parameters allow the user to specify device-dependent information. For simple devices, as in this case, the only user-alterable attribute is the speed at which the devices

operate. (A question from the user in response to the prompt ">>" will elicit the parameters the user may alter.)

We specify the SPEED of the paper tape reader to be 1000 characters per second. In this manner, we are able to significantly shorten the time needed to read the paper tape but only with the knowledge that in this application the bootstrap program's operation is independent of the speed of the paper tape reader. If we do not set the speed, PRIM will automatically select the actual value for each device. Though installation may only be done prior to execution, device parameters may be altered at any time via the SET commmand.

After installing the 1532, we must attach a source of input to the paper tape reader. This is accomplished by the PRIM MOUNT command. (The punch and TTY need not be mounted until needed.)

```
>MO<esc>UNT (A,I,N,OL,OU,T,?) ? One of the following:
 APPEND
 INPUT
 NEW
 OLD
 OUTPUT
 THIS-TERMINAL
>MOUNT (A,I,N,OL,OU,T,?) I<esc>NPUT (from file) TEST.ABS;2<esc> (on device)
 R<esc>EADER-1532 <cr>
>>? BINARY or ASCII
>>B<esc>INARY (with byte size) <esc>36
>><cr>
```

The second word of the command (in this case INPUT) determines the direction of IO and whether an existing or new file name is being specified (in this case, we wish to read from an existing file; for the paper tape reader, only INPUT and THIS-TERMINAL are meaningful). Following the file type, we supply the file name, TEST.ABS, followed by the name of the device on which to mount the file -- READER-1532.

Next we are prompted with ">>" for more information, in this case the format of the data on the file attached to the device. BINARY N implies the data file is to be treated as a stream of characters in which all eight bits of data are significant. The bytesize, N, describes the packing of the information in the file. For disk files written by EMLOAD, the UYK-20 loader, each byte of data occupies the low order eight of thirty-six bits -- hence the bytesize is specified as 36. ASCII specifies a standard 7-level text file; the bit corresponding to the eighth column (parity) will always be zero. If neither BINARY nor ASCII is entered, the default (BINARY 36) will be used.

Once we have installed the 1532 and mounted a file on the reader, we may begin the bootstrap.

```
>D esc EBUG
#Set STOP1,STOP2,BOOT2,LOAD cr
#Go (to) cr
--> Halted from 066: 00
--> AN/UYK-20 halted at 070, Used 0:00.4
#Return (to EXEC) cr
>SA esc VE M esc EMORY (on file) TEST.MEM;1 esc cr
>D esc EBUG
#Go (to) 01000 cr
```

After we have installed the device and mounted a file on it, we enter the debugger and set switches BOOT2, STOP1, STOP2, and LOAD, which correspond to similar switches on the UYK-20. (The momentary action of LOAD is accomplished by having the emulator clear the switch after responding to it, but note that the load does not begin until a Go command is entered.)

Eventually the emulator halts after reading the paper tape. Since the program is large, and we intend to re-execute it in future sessions, we save the core image after loading on a new file, TEST.MEM, where it can subsequently be retrieved via the RESTORE command.

The remaining examples will be much briefer than those presented above. Instead of complete sequences of interaction with an actual program, just those commands that are necessary to solve particular problems will be shown. In a few cases intervening interactions have been edited out of the transcript to emphasize the essential commands. In general, results will not be shown.

FIND WHICH INSTRUCTIONS MODIFY A LOCATION

A typical debugging problem is finding which instructions are changing a location (e.g., some module is clobbering a cell). This can be solved very easily with a simple break-time debugger program which traces all modifications of that cell. If the contents of location 012345 are being changed improperly, the following breakpoint command will identify all subsequent modifications of the cell, allowing the user to verify their validity.

```
#Break (at) 012345 esc (after doing) Write esc
 ##Mode Instruction esc ###Type @PCOLD cr
 ##Type 012345 cr
 ##Go (to) cr
 ## cr
<Program number is [2]>  #
```

The debugger commands following the "##" prompt are saved as the break program associated with the writing of 012345. After every write reference to that location, UYK-20 execution will be suspended and control will be passed to the debugger, which will execute the break-time commands. When the UYK-20 breaks, PC has already been advanced to the next instruction; PCOLD still addresses the previous instruction. Again, the contents-of operator is used to access the instruction location in memory. Since there is a Go command in the program, UYK-20 execution continues automatically after each break, thus generating a trace of the writes. If only the first breakpoint subcommand had been entered (eliminating the type-out of 012345 and the go commands on the third and

fourth lines), then after displaying the instruction, the debugger would display location 012345 and stop the UYK-20 (thereby returning to user command level).

FIND WHICH INSTRUCTION SETS A LOCATION TO A VALUE

A related, and perhaps even more common, problem is to find which instruction is setting a known (probably improper) value into some location. This can be accomplished with a variant of the break-time debugger program presented above.

```
#Break (at) 012345esc (after doing) Write esc
 ##If @012345 <> 067esc <then> ###Go (to) cr
 ##Mode Instruction esc ###Type @PCOLDcr
 ##cr
<Program number is [3]>   #
```

The first command in this break program is a conditional command, stating "If the contents (@) of 012345 are not equal (<>) to the value 067, then continue UYK-20 execution (Go)." When the Go is executed, the remainder of the break program is ignored and UYK-20 execution resumes immediately. When the given value is found, the Go is not executed, the break program is completed, and control returns to the debugger. Unlike the program above, this program produces no output until the tested value is found in the cell. (Should we forget the contents-of operator, then the value 012345 would be compared to the value 067 at each reference, and we would always resume execution.)

DETERMINE HOW MANY TIMES A CODE SEQUENCE IS ENTERED

Occasionally the operation of a program system is degraded by unnecessary and unexpected calls on subroutines that do initialization or other operations whose repetition do not cause errors but do affect performance. The PRIM Debugger can be used to count the number of times a code sequence is entered.

In the following example the code sequence is assumed to start at location 01234, and location 0100 (which is assumed to be unused) is used as a counter.

```
#Clear 0100cr
#Break (at) 01234esc (after doing) Xecute esc
 ##Set 0100esc = @0100+1cr
 ##Type 0100cr
 ##Go (to) cr
 ##cr
<Program number is [4]>   #
```

Every time location 01234 is entered, the count will be incremented and displayed. If only a final count is desired, rather than a running count of each execution, the following command could be used:

```
#Clear 0100cr
#Break (at) 01234esc (after doing) Xecute esc
 ##Set 0100esc = 0100+1cr
 ##Go (to) cr
 ##cr
<Program number is [5]>   #
 ...
#Type 0100cr
```

COUNT REFERENCES PRIOR TO A DESIGNATED CONDITION

The efficiency of a process can sometimes be evaluated by the number of times a data location is referenced prior to the occurrence of a given condition of interest. A variant of the previous example can be used where the automatic continuation is conditional on the designated condition not yet having occurred.

```
#Clear 0100cr
#Break (at) 012345esc (after doing) Read Write esc
 ##Set 0100esc = @0100+1cr
 ##If @05432 <> 0esc <then> ###Go (to) cr
 ##cr
<Program number is [6]> #
 ...
#Type 0100cr
```

When the condition that location 05432 contains a zero occurs, the program execution will break and the counter can be examined.

TRACE A LOOP ONLY ONCE

With the PRIM Debugger, a program trace is accomplished by setting an execute break on all instructions of interest and supplying a break-time debugger program that displays the most recent instruction executed (see the detailed example on entering and debugging a small program). To trace a loop only once, the continuation can be made conditional on *PC* not being equal to the starting location of the trace.

```
#Break (at) 01234:02345esc (after doing) Xecute esc
 ##Mode Instruction esc ###Type @PCOLDcr
 ##If @PC <> 01234esc <then> ###Go (to) cr
 ##cr
<Program number is [7]> #
```

The difference between this example and a single-stepped trace (using the .STEP breakpoint) is that here any called routines are not traced. To trace the loop *n* times, the continue could be made conditional on a counter that is incremented whenever the starting location is reentered.

DETERMINE WHICH INSTRUCTIONS WERE *NOT* EXECUTED

An interesting use of the PRIM debugger is in the isolation of those instructions within some region which were *never* executed while running some program. This can be accomplished by setting breakpoints throughout the area of interest, and then having each execute-break remove its own breakpoint.

```
#Debreak (from) esc all [confirm]cr
#Break (at) 01234:05670esc (after doing) Xecute esc
 ##Debreak (from) @PCOLD:+(IR**200X)>0**(IR--0F000X)<>0C000Xcr
 ##Go (to) cr
 ##cr
<Program number is [8]>
#Break (at) 05672esc (after doing) Xecute cr
#Go (to) 01234cr
 ...
--> Break after executing 05672: J 010000   #Break (at) cr
03457-04557 <X>[8]   05672 <X>   #
```

As each instruction breaks, its own breakpoint is cleared by the Debreak command. *IR* is the instruction register, containing a copy of the (first word of) the most recent instruction executed; *@PCOLD* is the address of that instruction. The long expression following :+ evaluates to 0 or 1 according to whether the instruction is short or long, thus clearing the breakpoint at both words of a long instruction (the second term is required only if RL instructions are used). At the end of the program, a display of the remaining breakpoints shows those locations that were never executed.

DETERMINE WHEN DATA CHANGE OVER A CODE SEQUENCE

It is occasionally necessary to determine whether a code sequence has changed the value in some location. This can be done by setting a breakpoint at the beginning of the sequence, where the break-time commands copy the data value into an unused location, and setting another breakpoint at the end of the sequence, where the break-time commands compare the copied value with the current value.

In the following example, the code sequence starts at 012345 and ends at 023456, the critical location is at 05432, and 0100 (assumed here to be unused) is used for temporary storage.

```
#Break (at) 012345esc (after doing) Xecute esc
 ##Set 0100esc = @05432cr
 ##Go (to) cr
 ##cr
<Program number is [9]>
#Break (at) 023456esc (after doing) Xecute esc
 ##If @05432=@0100esc <then> ###Go (to) cr
 ##cr
<Program number is [10]>   #
```

The first breakpoint, at the entry to our routine, saves a copy of the contents of 05432; the second breakpoint, at its exit, compares the current contents with that saved in 0100. Each time the value in 05432 changes over the designated code sequence, program execution will break. Should we be skeptical about the availability of location 0100, we could establish a breakpoint there which would break on any reference.

SEARCH A BUFFER FOR A GIVEN VALUE

The final example will show how to search a buffer (or any arbitrary set of locations) for the occurrence (or non-occurrence) of a designated value. In this example, all words containing 4 in the left half are located.

#*L*ocate *400X*[esc] (with mask) *0FF00X*[esc] (in) *013500:+3*[cr]
#

The general form of the locate command requires a comparison value, a mask, and a set of addresses to examine. The comparison value and mask can each be any arbitrary expression. The set of addresses can be a list of discrete locations or address ranges. By entering *NON* before the comparison value, the test is inverted. The comparison value defaults to "NON 0" and the mask defaults to "NOT 0" (i.e., all 1 bits). The test is identical to a masked compare (CM) instruction with the normal form being a test for equality and the NON form a test for inequality. Each location in the list is then displayed or not according to whether a match or non-match was specified. The locate command is analogous to the type command in that if it is terminated by an *escape* a replacement value can be entered for each displayed location. The same rules for the replacement value apply for locate-with-replacement as for type-with-replacement.

APPENDIX: UYK-20 REFERENCE LISTING

DEBUGGER NUMBERS AND OPERATORS

The debugger evaluates user input expressions in 16 bit twos complement arithmetic identical to that of the UYK-20. The operands of expressions include symbols, numbers, and character constants; the operators are listed below, and include all those found in the ULTRA assembler. Expressions may be parenthesized to an arbitrary depth.

Input numbers may be octal (leading 0), decimal (leading non-0), or hexadecimal (leading digit and trailing X). Output is identical, except that when hexadecimal output is requested, a leading A thru F is not prefixed with a 0. A character constant consists of one or two characters included in single quotes; the ASCII character set is used, allowing 8 bits per character.

The integer arithmetic operators are +, -, *, /, MOD, and ABS. (MOD is the remainder after division, while ABS is a unary operator returning the absolute value of the following term.)

The relational operators are =, <, >, <=, >=, and <>. Their value is 1 when the relationship is true and 0 otherwise.

The Boolean operators are ++ (OR), -- (XOR) and ** (AND).

The fetch operator is @. @ is a unary operator whose value is the contents of the cell addressed by the next term. @ is invaluable when building conditional expressions in break-time programs. It also provides a good shorthand at other times, e.g., "T(ype) @PC" displays the next instruction to be executed

UYK-20 PARAMETERS AND DEVICES

Various UYK-20 options are selectable as parameters of the emulated UYK-20. The settings of these parameters may be inspected and changed by the user at any time via the SET and SHOW commands.

CLOCK sets the clock source frequency, expressed in ticks per second (Hz). The default value is the internal source frequency of 1000 Hz.

MEMORY sets the size of memory, expressed in 8K modules. The default is a half memory of 4 modules.

NDRO selects the particular NDRO program, by name. The selections available are listed in response to the Set subcommand *NDRO ?*. The default is the Standard NDRO.

REGISTER-SETS controls the installation of the optional register set. The parameter value is either one or two sets; the default is one. When only one set is installed, the general-register-set selector bit in status register #1 is ignored.

Devices are installed on UYK-20 channels via the INSTALL command. Installation of a device implies the simultaneous installation of the appropriate type of channel. Therefore the user need never be concerned with the installation of channels. Furthermore, all channels are available for the installation of any type of device, without restrictions concerning channel groups; also, 32-bit channels do not require the use of a second channel (at n+1 or n+4) to carry half the data. It is therefore possible, though not necessarily useful, to specify configurations which cannot be built on an actual UYK-20. The list of implemented devices that may be installed is listed in response to the command *INSTALL ?*.

One optional parameter which is specifiable for every installed device is the device speed, expressed in some units appropriate to the device (characters per second, cards per minute, inches per second, ...). The default is either the actual speed or a typical speed of that device. Altering the parameter linearly alters all the timing associated with that device.

Temporary speed increases can be useful in speeding up lengthy processes for which timing is not critical. The principal example is a bootstrap load; a factor of two or four increase in the emulated speed of the bootstrap device results in a substantially faster bootstrap. Speed changes may also be used to experiment with the effect of variations in data rate on program performance or throughtput.

For the device TTY-1532 (the TTY on the 1532 operator console), interrupt transmission of the next character is indicated by entering the two-character sequence *control-shift 0*, where *control-shift* is the PRIM control-shift input control code (not the control-shift key on the terminal.)

UYK-20 SYMBOLS AND CELLS

In addition to memory, the following cells are known to the debugger and are accessible by the indicated names. Except as indicated, modification of any cell affects future behavior of the UYK-20 in the obvious manner. For example, changing the state of an I/O chain from idle to chaining will cause that chain to resume execution (at its chain address pointer location) when execution is next resumed. The only note of caution concerns the device-to-channel bits in the I/O interface; they may not be altered with confidence since they are inextricably tied to (inaccessible) device status information.

R0 thru *R15* (decimal): the primary register set.

RR0 thru *RR15* (decimal): the secondary register set.

PG.0 thru *PG.77* (octal): the page registers.

N.0 thru *N.77*, *N.300* thru *N.477* (octal): NDRO memory. Any attempt to store into NDRO generates a break, and the write is ignored.

TRIG.0 thru *TRIG.17* (octal): The trigonometric CORDIC constants.

HYPER.0 thru *HYPER.17* (octal): The hyperbolic CORDIC constants.

The following are one-bit cells which correspond to control panel switches and clock enable functions.

> ***MCLEAR***: auto-start switch. When set by the user, causes the UYK-20 to execute a master clear sequence and then begin executing. MCLEAR is always cleared by the emulator at the end of the master clear. The master clear sequence includes a reset of all the emulated devices as well as the regular CPU and IOC reset.
>
> ***LOAD***: load switch. Causes the UYK-20 to execute a master clear sequence, set the PC to 2, and begin executing. LOAD is cleared by the emulator at the end of the master clear.
>
> ***BOOT2***: Bootstrap 1/2 selector. Set is bootstrap 2; clear is bootstrap 1. Never modified by the emulator.
>
> ***STOP1, STOP2***: Set is ON; clear is OFF. Never modified by the emulator.
>
> ***RTCE***: Real time clock enable, allows the clock to be advanced at each tick interval.
>
> ***MCRE***: Monitor clock enable, allows the clock to be decremented at each tick interval.
>
> ***RTCIE***: Real time clock interrupt enable, allows the real time clock overflow interrupt request to be generated.

The following are the miscellaneous other registers accessible to the user.

> ***PC***: the current program counter (containing address of the next instruction).
>
> ***SR1, SR2***: the status registers.
>
> ***RTCU, RTCL***: the real time clock, upper and lower.
>
> ***MCR***: monitor clock register.

The following cells contain bit-encoded status information; in each case a set (1) bit indicates either a pending interrupt or an enabled condition. For all but the first, one bit is used for each channel, with bit 0 for channel 0, ..., and bit 15 for channel 15.

> ***INT1,2***: pending class I and II interrupts.
>
> Bit 15: Power fault (never generated internally).
> Bit 14: Memory resume. (Generated internally only when a reference is made to nonexistent memory.)
> Bit 13-8: Class I, priority 3 thru 8.
>
> Bit 7: CP Instruction fault.
> Bit 6: IOC Instruction fault.
> Bit 5: Floating point overflow/underflow.
> Bit 4: Executive return.
> Bit 3: RTC overflow.

Bit 2: Monitor clock overflow.
Bits 1-0: Class II, priority 7 thru 8.

INT3IC: Class III intercomputer timeout interrupts pending.

INT3X: Class III external interrupts pending. (This bit is set at the time that the status word is stored in memory.)

INT3O: Class III output chain interrupts pending.

INT3I: Class III input chain interrupts pending.

EIE: Channel external interrupt enable flags.

CHIE: Channel interrupt enable flags.

The next five items are merely informative; altering them does not affect the future behavior of the UYK-20.

IR: Instruction register containing the last instruction executed (the first word for a long instruction).

EA: Effective address of the instruction in IR. For RK, it is the effective constant. (Displaying IR in instruction mode will display both IR and EA for long instructions.)

CAP: Channel address pointer, indicating the type of the last instruction cycle executed. Zero indicates CPU (DPS) execution; a value of 8n+2 indicates IOC execution by the input chain on channel n; a value of 8n+6 indicates the output chain on channel n.

MAR: Memory address register, giving the last relative memory location referenced. If the last reference was an instruction fetch, MAR addresses the location following the instruction. MAR is set by channel transfer cycles and interrupt cycles as well as by execution cycles.

PCOLD: the program counter at the start of the last CPU instruction cycle.

The remaining items are all concerned with the UYK-20 channels and channel-device interfaces. Hexadecimal notation is used to designate the channel number with a single digit of 0 thru F.

CM.0 thru *CM.FF* (hexadecimal): channel control memory. The two-digit number designates the channel in the first digit and the word-within-channel in the second digit.

ICH.0 thru *ICH.F*: the input chain state for each channel.

OCH.0 thru *OCH.F*: the output chain state for each channel. For both input and output chains, the state values are:

0: Idle.
1: Chaining.

2: Search for first sync (synchronous communication channel only).
3: Search for second sync.
4: Input data transfer.
5: Output data transfer.
6: Function transfer (parallel and NTDS channel only).
7: Forced function transfer (parallel and NTDS channel only).

All channel-device data transfers take place identically in the emulator, regardless of the actual channel type. Transfers occur one byte (8, 16, or 32 bits) at a time, following the model defined for a parallel I/O channel, but with a buffer to hold each datum in turn. Each transfer uses the appropriate buffer for the channel and two interface bits, a request bit and an acknowledge bit. For parallel and NTDS serial channels, functions are transferred the same way, while status uses EIE and EIR.

IB.0 thru *IB.F*: the input (device to channel) buffers.
OB.0 thru *OB.F*: the output (channel to device) buffers.

IDR.0 thru *IDR.F*: input data request (device to channel) flag.
IDA.0 thru *IDA.F*: Input data acknowledge (channel to device) flag.
ODR.0 thru *ODR.F*: Output data request (device to channel) flag.
ODA.0 thru *ODA.F*: Output data acknowledge (channel to device) flag.
EFR.0 thru *EFR.F*: Function request flag (parallel and NTDS serial).
EFA.0 thru *EFA.F*: Function acknowledge flag (parallel and NTDS serial).
EIR.0 thru *EIR.F*: External interrupt request flag (parallel and NTDS serial).

The static lines for communication channels are given their MIL-STD-188 names (A thru L); A, D, F, G, H, and J are the outbound lines, while B, C, E, I, K, and L are the inbound lines. These flags are not valid for parallel or NTDS serial channels. The eight RS-232C control lines are mapped into these lines as noted in the list below; the mapping is taken from the channel cable pin assignments.

IOA.0 thru *IOA.F*: Loop test.
IOB.0 thru *IOB.F*: Ring indicator (CE).
IOC.0 thru *IOC.F*: Carrier detect (CF).
IOD.0 thru *IOD.F*: Data terminal ready (CD).
IOE.0 thru *IOE.F*: Clear to send (CB).
IOF.0 thru *IOF.F*: New sync.
IOG.0 thru *IOG.F*: Request to send (CA).
IOH.0 thru *IOH.F*:
IOI.0 thru *IOI.F*:
IOJ.0 thru *IOJ.F*:
IOK.0 thru *IOK.F*: Data set ready (CC).
IOL.0 thru *IOL.F*:

BREAKPOINTS

The UYK-20 emulator is continually monitoring the execution of the UYK-20 to detect the occurrence of any break conditions which may be set. (Breaks are neither set nor cleared by the emulator; only the user, via the debugger breakpoint commands, can set or clear breaks.) When a break condition is detected, the break occurs after the event, at

the end of the current cycle of execution. It is therefore possible for multiple breaks to be reported at one time. Each UYK-20 cycle consists of the execution of one of the following items:

1. A CPU instruction, including indirect addressing (for the IOCR or REX instruction, execution of the IOC command or remote instruction takes place in the same cycle).

2. An IOC chain instruction.

3. A CPU interrupt sequence due to a pending interrupt request. This cycle includes the storing of PC, status and clock, and the fetching of new PC and status. The creation of a pending interrupt is a side effect of a previous cycle (or a manual operation by the user).

4. A single channel data transfer under control of a buffer control word.

5. A clock cycle, which occurs at each clock tick interval whether or not the real-time and monitor clocks are enabled.

6. A device execution cycle.

7. An external interrupt cycle, in which a channel stores a status word and generates an external interrupt request.

There is an event associated with each of the first five types of cycles, allowing the user to break execution after any cycle of the given type. Devices cannot be directly monitored; one must monitor the channels instead. The external interrupt cycle can be monitored via a write break set at any or all of the EI Storage locations in memory.

The following event flags are defined for the UYK-20:

.STEP: Break after any CPU execution cycle.

.CHAIN: Break after any chain or command execution.

.INT: Break after any interrupt cycle.

.XFR: Break after any IOC transfer cycle.

.TICK: Break after any clock tick interval.

.JUMP: Break after any transfer of CPU control to other than the next instruction. Includes interrupts as well as (successful) conditional jumps, but does not include NOP (LJ *+1).

.STORE: Break after any memory store, including those generated by I/O and interrupts.

.NJMP: Break after any CPU conditional jump instruction which doesn't jump.

.ANOM: Break after any of a number of switchable anomalies, including:

Illegal Instruction. An illegal CPU or IOC instruction is executed. The instruction has set an interrupt request in *INT1.2*, which will cause an interrupt when execution next resumes - unless it is cleared.

External Interrupt while not enabled. A device (on a parallel channel) generated an external interrupt request when EIE was not set on that channel.

Improper Indirect Address Word. The J value is one of the unassigned values.

An IO transfer specifying EF or EFF is executed on a communications type (not NTDS) serial channel (the EF/EFF is turned into a NOP).

A communications channel operation (such as CSIR or CSST) is executed on a parallel channel (the operation is turned into a NOP).

In addition, the following anomalies always generate a break, regardless of the setting of .ANOM:

Device error. Usually occurs when an unconfigured channel executes an EFF.

Odd memory address (in a double word reference). The addressed word has been used twice.

Odd register number (in a double register reference). The addressed register has been used twice.

Store into NDRO. The store is not performed.

In addition to the above events, breaks can be set at specific locations, to occur only when the location is referenced in the indicated manner. The three forms of reference are:

X: Execute includes the fetch of a short instruction, or the fetch of the first word of a long instruction. Applies to CPU and IOC execution equally.

R: Read includes any other fetch. (Neither break includes the second word of a long instruction.)

W: Write includes any store into the given cell for any reason.

These reference breaks may be applied to the following cells:

Memory: all three forms, in any combination, by absolute (not relative) address.

NDRO: Read and Execute breaks.

Registers: Write break, in each set separately.

Channel control memory: Write break, on each pair of words. (A break applies to both words of an even-odd pair regardless of the cell actually specified.) A break on a channel address pointer can be used to single step a particular chain, while a break on a buffer control word can be used to single step a particular transfer.

Inp[illegible] [illegible]d Output (Interface) Buffers: Write break, for each separately. The break is always taken after the cycle which wrote the word, but before the cycle which will read th[illegible] word. Therefore, any alteration of the data is effective. The break on the input buffer is the only break triggered by a device cycle (other than a device error anomaly).

PRIM SYSTEM: USER REFERENCE MANUAL

INTRODUCTION

This document is the common reference manual for all users of the PRIM system, both those using one of the existing emulation tools and those writing new emulators. For the former, this manual is supplemented by the appropriate tool-specific guide (*e.g.*, *PRIM System: U1050 User Guide*); for the emulator writer, the supplement is *PRIM System: Tool Builder Manual*.

The PRIM system is always in one of three states, known as the exec, the debugger, and the target execution states. The transition between states is controlled by the user. Both of the first two states are PRIM command processors that take commands from the user and execute them. The exec, whose command prompt character is ">", is used principally for setting up a target environment; the debugger, whose command prompt is "#", is used for the detailed examination and control of the executing target machine. Target execution includes the emulation of not only the CPU, but also clocks and assorted peripheral IO devices. The three sections following the introduction describe each of the states in turn.

The PRIM exec and debugger commands are illustrated with examples taken from actual session transcripts. In all the examples, user input is ***italicized*** to distinguish it from PRIM output. Input control characters appear as their abbreviations superscripted (*e.g.*, ^{esc}).

GENERAL INPUT CONVENTIONS

User input to PRIM, both exec and debugger, is generally free-format and case-independent. Leading spaces and tabs are ignored, and lower case is treated as its upper case equivalent (except in quoted strings, where case is potentially significant). User input to the target machine during target execution state is in the format required by the target system.

Certain characters have been assigned editing and intervention functions when input by the user. The editing characters apply only to the PRIM exec and debugger, while the intervention characters apply to the target execution state as well. The specific characters assigned to most of the functions may be altered (via the exec Change command) to suit one's needs. The editing functions are valid at any time during PRIM command input; commands are not executed until after the final character has been accepted.

> ***Back-space*** (cntl-H) erases a character from the current word or term of input. The back-space is echoed as a backslash (\) followed by the erased character. When there are no erasable characters, a bell (cntl-G) is echoed instead.
>
> ***Alternate back-space*** (initially cntl-A) performs a function identical to ***back-space***; it is provided as a convenience.

Backup (initially cntl-W) erases the current word or term of input. It is echoed as backslash (\) followed by the first character of the erased word.

Retype (initially cntl-R) retypes the current input line; it is useful after a confusing amount of editing has occurred.

Delete (initially DEL or RUBOUT) aborts the current input command or subcommand, allowing the user to re-enter it. It is echoed as " XXX".

Question (?), when entered at the beginning of a command field, elicits a description of the expected input, followed by a retype of the line. When the expected input is a selection from a list (or menu), the entire list is shown.

The intervention characters are valid at any time, including command input, command interpretation, and target execution.

Abort (initially cntl-X) interrupts the current activity and returns control to the command level of either exec or debugger. When used to cancel an exec or debugger command, control returns to the top level of the same state; abort is the only means of canceling a command when the user is in subcommand mode. When used to interrupt target execution, control returns to the state from which execution was initiated; abort is the only means of stopping a looping target machine.

Status (initially cntl-S) produces a one-line summary of target machine status, including program counter, emulated elapsed time, and active IO devices. The command is valid at any time, but useful primarily in execution state.

The following character is active only during target execution.

Control-shift (initially cntl-1) permits the user to enter (during execution) a control code that cannot be entered directly because it is intercepted by either PRIM or the operating system; the PRIM characters inolved are ***status***, ***abort***, and ***control-shift*** itself. The next ASCII character following the ***control-shift*** (other than the digits 0 thru 9) has its two leading bits cleared, thus converting it to an ASCII control code (***A*** or ***a*** to ***cntl-A***, etc.). *Control-shift* followed by a digit results in an input that is outside the normal target character set and is used for particular target-machine-dependent functions. The control-shift character itself is not echoed, and not passed to the target machine. If execution terminates before that next character is input to the target device, the control-shift is canceled; it is not retained for the next resumption of execution.

PRIM EXEC

The PRIM exec is the initial state of a PRIM session. Exec commands are concerned primarily with building target configurations, saving PRIM session results, restoring previously saved sessions, and accessing or creating files (within the file space of the host operating system).

The exec prompt character is ">", indicating that PRIM is in exec state and that the exec is awaiting a new command; it is always shown on a new line. Individual input fields consist of keywords (a word selected from a menu), decimal numbers, and file names. Exec commands are composed of fixed sequences of fields, each terminated by a delimiter character; a final confirmation consisting of a ***return*** is often required.

Keywords are selected by any unambiguous leading substring. Often, a single character suffices; three characters are always sufficient. Numbers are specified in their entirety. File names are specified according to the conventions of the operating system. All commands that will use a file for output require the name of a new file (except the Mount-Append and Mount-Old commands, which modify existing files); all other file commands require the name of an existing file. In TENEX, an existing file name -- and a new file that is a new version of an existing file name -- is recognized (and completed) in response to an input ***escape***.

The normal delimiters that terminate command fields are ***return***, ***escape***, and ***space***. ***Escape*** and ***space*** function identically except that the former generates feedback to the user while the latter generates none; the feedback produced by ***escape*** includes both field completion and next-field prompting (which is given in parentheses). ***Return*** is used to complete a command immediately, bypassing any remaining fields and confirmation; if further input is required, the ***return*** is treated as an ***escape***. (In the examples that follow, ***escape*** termination is used to show the prompts.)

Keywords that involve either devices or parameters are machine-dependent; the selections shown in the examples are meant to be illustrative rather than definitive. Device specification is further complicated when two (or more) of the same generic device are installed. Therefore, for device names, two further delimiters are utilized, ***at*** ("@") and ***colon*** (":"). A fully qualified device name consists of ***generic-name @ channel-number : unit-number***; the numbers are required only to the extent necessary to specify a particular device. When a device name is terminated by one of the standard terminators, and when further disambiguation is required, the exec prompts explicitly regardless of the terminator.

The remainder of this section consists of the descriptions of the exec commands in alphabetical order. Each command description begins with a transcript showing one or more examples of the command and its various options. Those commands that require a second keyword show that list via an input ***question***. The exec commands are:

```
>? One of the following:
 CANCEL
 CHANGE
 CLOSE
 COMMANDS
 DEBUG
 FILESTATUS
 GO
 INSTALL
 MOUNT
 NEWS
 PERIPHERALS
 QUIT
 REASSIGN
 RESTORE
 REWIND
 SAVE
 SET
 SHOW
 SYMBOLS
 TIME
 TRANSCRIPT
 UNINSTALL
 UNMOUNT
>
```

<u>Comment.</u>

>; *this line is a comment*[cr]
>

Any line beginning with a *semicolon* is treated as a comment. Comments are recorded in the transcript if one is open (see Transcript command).

<u>Cancel</u> abandons all outstanding IO operations for a designated device.

>*ca*[esc]NCEL (IO for device) *ta*[esc]PE-UNIT [cr]
>

This command is intended for use when, after an IO error halt (described in the section on target execution), the user wishes to abandon the device operation rather than mount a file and retry the operation. The list of outstanding IO operations, by device, is part of the Peripherals command output.

Change reassigns the PRIM control functions.

```
>ch^esc^ANGE (input code for) ? One of the following:
 ABORT
 ALT-BACKSPACE
 BACKUP
 DELETE
 RETYPE
 STATUS
 CONTROL-SHIFT
>CHANGE (input code for) ab^esc^ORT (from ↑X to) ? A Control Code.
>CHANGE (input code for) ABORT (from ↑X to) ↑P cr

>ch^esc^ANGE (input code for) d^esc^ELETE (from <DEL> to) ^esc^ [not changed]
>
```

This command allows the user to change the ASCII control code assigned to any of the listed PRIM control functions from its current assignment to another (currently unassigned) control character. The function name is the second word of the command; when it is terminated with an *escape*, the current assignment is noted in the noise. The entire set of ASCII control codes (including *delete*) is available excepting *null*, *back-space*, *line-feed*, *return*, *escape*, and *unit-separator* (TENEX *end-of-line*) which have fixed functions in PRIM. For *abort* and *status* the set is limited to *cntl-A* thru *cntl-Z*.

Close terminates the current transcript file if one is open.

```
>cl^esc^OSE (transcript file.) cr
>
```

A transcript file is opened using the Transcript command; it is automatically closed at the end of a session.

Commands redirects subsequent input from a file.

```
>co^esc^MMANDS (from file) command.file^esc^ cr
>
```

This command causes PRIM to read its subsequent command input from the named file instead of the user terminal (or current command file). The file input is treated exactly as terminal input except that intervention functions (*abort* and *status*) are valid only from the terminal. Should a command in the file cause execution to be resumed, input that normally would come from the user terminal is taken instead from the file. Input reverts to the previous source at the end of the file; an *abort* terminates all command files and reverts input to the user terminal. Command files may be nested. Command files are very useful for common session-initialization sequences.

Debug transfers control to the PRIM debugger.

```
>d^esc^EBUG
#return (to EXEC) cr
>
```

The PRIM debugger is described in the next section; control is returned to the Exec via the debug Return command.

Filestatus returns information about mounted files for all or designated devices.

```
>f^esc ILESTATUS (for device) ^esc ALL
 Record    File Name          Device
 12        CARD.DECK          CARD-READER
 12        User Tty           PRINTER
 825       TERMINAL.INPUT     TERMINAL (In)
 12345     TERM.OUT           TERMINAL (Out)
 2+6       ABCD.EFG           TAPE-UNIT:0

>f^esc ILESTATUS (for device) ca^esc RD-READER
 Record    Type  Byte/Last     File Name
 12        Bin12 960/1280      CARD.DECK
>
```

When the device field is empty (*return* or *escape*) all mounted files are listed; otherwise just the file(s) on the named device are listed. The latter case gives more complete status than does the former. The output fields are:

> Record tells the current position of the device or the number of records which have been processed. For disks, it is a sector number; for card readers and punches, a card count; for communication lines, the total number of bytes transferred; for mag tape units, the position from beginning of tape expressed as files + records.
>
> File Name is the name of the file; the name "User Tty" is displayed when *THIS-TERMINAL* is the file.
>
> Device is the emulated device on which the file is mounted.
>
> Type describes the type of file, either Ascii or Binxx, where xx is the file byte size. The type may have been explicitly specified at mount time, or it may have been assumed by PRIM.
>
> Byte/Last is, for a mounted disk file, the current byte position in the file and the total number of bytes in the file.

The marginal notation "[not opened]" indicates that the named file could not be found (this occurs only to a restored file) and that the device must be reassigned to another file (or to the same file via a new path name).

Go transfers control to the target execution state.

```
>g^esc O (from 1234) ^cr
--> MACHINE running at 567A, Used 0:00.4
--> MACHINE halted at 6543, Used 0:01.0
>
```

This command transfers control from the PRIM exec to the emulator or target machine, in its current state. Control returns to the exec when the target machine halts or a breakpoint is encountered (see the debugger Break command) or the user interrupts execution with an *abort*.

In the example, the user followed the command with a *status* request (the *status* character itself is not echoed) resulting in the first reply line (MACHINE running at ...); the target machine is still running. Eventually the target machine halted, producing the second status line and returning control to the exec as evidenced by the exec prompt.

<u>Install</u> adds a designated type of device to the machine configuration.

```
>iescNSTALL (device) ? One of the following:
 CARD-READER
 PRINTER
 TAPE-CONTROLLER
 TERMINAL
>INSTALL (device) pescRINTER (CHANNEL) 1esc
>>? SPEED
>>sescPEED (characters per second) esc300
>>cr

>iescNSTALL (device) taescPE-CONTROLLER (CHANNEL) 3esc cr
How many TAPE-UNIT's do you want? 2cr
For the first TAPE-UNIT, (UNIT) 0esc cr
>>cr
For the second TAPE-UNIT, (UNIT) 1cr
>>cr
>
```

The device type is selected from among those implemented. The user is prompted for each necessary item of information, typically including an address for the device in the target IO address space and the number of units to install. After the required information is gathered, sub-command mode (">>" prompt) is entered to gather optional parameters; any optional parameter not supplied takes on its default value. Subcommands are terminated by an empty command, ***return*** only. An installed device is initially unmounted -- there is no file associated with the device for purposes of actual IO.

When the device being installed is a multi-unit controller, the dialogue proceeds through each of the individual units to gather their parameters. After the command is completed, the controller is no longer visible; only the individual units are. An ***abort*** aborts the entire command, not just the current unit.

Installation is permitted only before any execution has taken place. Typically, a user or user group installs a standard configuration and then saves it for use in all subsequent sessions (see the Save-Configuration and Restore commands). The optional parameters of an installed device may be changed at any time using the Set command.

<u>Mount</u> associates a file with an installed device.

```
>m^esc OUNT (A,I,N,OL,OU,T,?) ? One of the following:
 APPEND
 INPUT
 NEW
 OLD
 OUTPUT
 THIS-TERMINAL
>MOUNT (A,I,N,OL,OU,T,?) t^esc HIS-TERMINAL  (on device) p^esc RINTER cr

>m^esc OUNT (A,I,N,OL,OU,T,?) n^esc EW (in & out file) ABCD.EFG;1^esc (on device)
 ta^esc PE-UNIT cr
>

>m i^esc NPUT (from file) card.deck^esc (on device) ca^esc RD-READER cr
>>? BINARY or ASCII
>>b^esc INARY (with byte size) 12cr
>>cr
>
```

Associating a file with an installed device causes subsequent emulated IO for that device to be directed to the file. The second keyword following Mount determines the direction of data flow and the choice of an old (existing) or new file. A file must be mounted on a device before any actual IO can take place.

APPEND mounts an old file for output only, with the subsequent output being appended to the previous contents of the file.

INPUT mounts an old file for input only.

NEW mounts a new file for both input and output (the file is initially empty).

OLD mounts an old file for both input and output (subsequent output overwrites any existing file data).

OUT mounts a new file for output only. For a disk or tape device, OUT is treated as NEW.

THIS-TERMINAL associates the user terminal -- instead of a named file -- with the named device. The mounting is for both input and output unless a file has already been mounted for one, in which case the terminal is mounted only for the other. The terminal is known to be an ASCII "file". The terminal may be mounted only once for input; it may be mounted for output (or on an output-only device) any number of times, but the output is not labeled as to source.

Only some of the forms above are applicable to any given device. For a disk- or tape-like device, an INPUT, OLD, or NEW file is expected; an OLD file is one that was NEW in a previous PRIM session, and is being re-used, while an INPUT file is an old read-only file. For a bidirectional communication device (*e.g.*, a terminal), two files are required: an INPUT file and either an OUTPUT or APPEND file. Alternatively, a real terminal may be used for both (or either one). For an input-only device, INPUT and OLD are identical; for an output-only device, OUT and NEW are identical.

For those devices that deal exclusively with character data, the mounted file is always taken as an ASCII text file; character translation is performed as part of the IO process. (This allows the file to be created and/or processed by any operating system utility that deals with text files.) For tape and disk devices, the file format is internal to PRIM (and therefore not requested from the user); the data is recorded directly. For other devices the user is asked, via subcommand mode (">>" prompt), whether the mounted file (NOT the device) is an ASCII text file or a binary file containing a stream of pure data in bytes of some fixed size. The default is a binary file of a device dependent byte size.

Once a file has been mounted on a device, all exec commands that refer to the file require the device name as the specifier; for communication devices, where two files are normally mounted, the device name is followed by a direction selector. The file name itself is not used as the internal identifier.

News reads the PRIM on-line news file.

```
>n^esc EWS
 Do you want to see  4-APR-77  Changes in PRIM ?: esc YES
  [ Here comes the message regarding changes of 4-APR-77 ... ]
 Do you want to see 24-MAR-77  Preliminary Documentation ?: del XXX
>
```

The date of the most recent news message is shown automatically at the start of each session. In response to the command, each message's date and subject is shown, beginning with the most recent message. For each message, the body may be seen (*YES*) or skipped (*NO*), or the command may be terminated (*delete* or *abort*).

Peripherals returns information about the installed devices.

```
>p^esc ERIPHERALS
Chan  Unit  Mounted  Device
 1     0     No      PRINTER
 2     0     Yes     TERMINAL
 3     0     Yes     TAPE-UNIT
 3     1     Yes     TAPE-UNIT

 active devices: TERMINAL
>
```

This command produces a listing of all the installed devices, together with their IO addresses and a notation concerning whether they have files mounted. It also lists all devices which have suspended IO operations. Ordinarily, suspended operations are limited to (1) IO error conditions and (2) input operations where the input file is a real terminal and no input was available when target execution stopped.

Quit terminates a PRIM session.

```
>q^esc UIT
  Quitting MACHINE  [Confirm] cr
@
```

Terminating the PRIM session involves closing all open files and returning control to the process that initiated the PRIM session. The session cannot be continued.

Reassign specifies a new file for a mounted device.

```
>reaᴱˢᶜSSIGN (device) taᴱˢᶜPE-UNIT (to file) new.fileᴱˢᶜ cr
>
```

This command is used to substitute a new file specification when, after a prior Restore command, a previously mounted file cannot be found. In particular, a restore done from a different directory than the one in force at save time has trouble finding any of the mounted files. Reassign may only be used for devices/files that are marked "[not opened]" in a file status display. The new file is assumed to have the same characteristics as the old one and is positioned at the same file position.

Restore recovers the state information saved in a file.

```
>resᴱˢᶜTORE (from SAVE file) ABCD.CONFIG;1ᴱˢᶜ cr
  restored CONFIGURATION from TUESDAY, MAY 3, 1977 12:35:08 PDT
>
```

The current context is updated with the complete or partial environment previously saved in the designated file by the Save command. For the addressable regions -- machine memory, registers, etc. - the saved data replaces the current data only for those cells that were actually saved; cells not saved are not cleared. (Thus, nonoverlapping memory images are merged.) For nonaddressable regions -- symbol, configuration, and breakpoint -- each one is completely replaced if present in the file. The date and region(s) saved are shown, followed by a list of any mounted files that cannot be found.

Rewind returns a device's mounted file(s) to the beginning.

```
>rewᴱˢᶜIND (device) taᴱˢᶜPE-UNIT cr

>rew terᴱˢᶜMINAL (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>REW TERMINAL IᴱˢᶜNPUT cr
>
```

This command is useful for retrying a program without unmounting and remounting files. (Files are always rewound when mounted, except for Append files, which cannot be rewound.) For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be rewound; the default is *BOTH*.

Save copies selected state information into a file.

```
>saᴱˢᶜVE ? One of the following:
 ALL
 CONFIGURATION
 FORMATS
 MEMORY
 SYMBOLS
>SAVE cᴱˢᶜONFIGURATION (on file) aBCD.CONFIG;1cr
>
```

This command saves on the (new) file an image of the region(s) selected for saving. The contents of the file can later be restored for use in this or another session. The second word of the command selects one of the save options.

ALL saves everything -- a complete checkpoint of the target machine and debugging state. "Everything" includes memory, all addressable registers, installed devices, mounted files together with their positions, debug breakpoints and their programs, debug formats and modes, defined symbols, and the internal state of the emulated machine.

CONFIGURATION saves all the machine configuration data, including installed devices, mounted files (if any), machine parameters, and debug formats and modes. This command is allowed only before any execution takes place. Useful for creating a standard machine configuration (possibly with some standard files mounted) for use in subsequent sessions.

FORMATS saves all the formats that have been defined (using the debugger Format command).

MEMORY saves those regions of the machine memory that are not clear. (At the start of a PRIM session, memory is already cleared.)

SYMBOLS saves all the user-defined symbols, both those loaded via the exec Symbols command and those defined directly via the debugger New-symbols command. The file that results is a SAVE/RESTORE file, not a SYMBOLS file!

Set changes the values of user-settable parameters.

```
>se^escT (<empty> or device) ^cr
>>? One of the following:
 CLOCK
 MEMORY
 SPEED
>>c^escLOCK (ticks per second) ^esc1000 ^cr
>>m^escEMORY (8K modules) 4^cr
>>^cr

>se^escT (<empty> or device) p^escRINTER
>>s^escPEED (characters per second) 150^cr
>>^cr
>
```

Following the command word, the user selects the group of parameters he wishes to alter. An immediate *return* selects the global machine parameters; a device name selects the parameters of that particular installed device (the parameters of multiple installed instances of the same device type need not have identical settings).

Any number of parameters from the selected group may be changed. In response to the subcommand prompt (">>"), the name of a parameter and its new value are entered; each change is made immediately and a new subcommand prompt appears. The command is terminated by an empty input, *return* only, or by an *abort* (which does not undo any parameters previously changed). The list of possible parameters is highly machine and device-dependent; it typically includes the size of memory and the speed of each device.

The value of a parameter is either a (decimal) number or a keyword from a parameter-specific list; a *question* in the value field reveals which is expected. An *escape* sets the parameter to its default value.

Show displays the values of all the parameters in a group.

```
>sh^esc OW (<empty> or device) cr
 CLOCK is 1000 ticks per second
 MEMORY is 4 8K modules
 SPEED is 750 nanoseconds per memory cycle

>sh^esc OW (<empty> or device) p^esc RINTER
 SPEED is 200 characters per second
>
```

Following the command word, the user selects either the global machine parameters (*return*) or the parameters of an installed device. The names and current values of all the parameters are displayed.

Symbols reads an ASCII symbol-table file.

```
>sy^esc MBOLS (from file) SYMBOLS.EXAMPLE^esc cr
>
```

This command causes PRIM to build a user-defined symbol table from the data in the named file, which is a structured ASCII text file. The file may define values for both global symbols and program-local symbols that are organized into programs. In the PRIM debugger, the global symbols plus the local symbols of the currently open program are accessible at any time. Symbol values in the file are octal. The form "name == value" defines a global symbol; the form "name = value" defines a local symbol; the form "name:" establishes a program name to which subsequent local symbols are assigned. The file is free-format in that spaces, tabs, commas, and new-lines may occur anywhere -- except in the middle of names or values. The following is a sample symbols file.

```
ALPHA==45
BETA==12345
PRI: A=2000, B=2132, C = 2241
XYZ:
A=3212   AA=3245, AAA=3261,AAAA=7777
```

Symbol files are intended to support the moving of symbolic label data from an assembler or linking loader into PRIM for use in symbolic debugging.

Time displays time-of-day and time-used information.

```
>ti^esc ME (is) TUESDAY, MAY 3, 1977 12:34:33-PDT
 Used 0:14.6 PRIM time;  Used 0:02.7 MLP time.
>
```

This command displays the date, time of day, the amount of PRIM time used and the amount of MLP-900 time used in this PRIM session. (Elapsed target machine time is displayed in response to *status*.)

Transcript transcribes the subsequent PRIM session on a new file.

```
>tr^esc ANSCRIPT (to file) new.file^esc cr
>
```

All transactions with the user terminal, including execution-time IO to THIS-TERMINAL, is transcribed until either the user terminates the session (with a Quit command) or closes the transcript. Only one transcript may be open at a time. A header line containing the date and time is placed at the head of the file.

Uninstall removes an installed device.

```
>uni^escNSTALL (device) ? PRINTER or TAPE-UNIT
>UNINSTALL (device) ta^escPE-UNIT (unit): 1^esc ^cr
>
```

This command is the inverse of the Install command; it removes an installed device from the configuration, first unmounting its files if necessary.

Unmount unmounts the file(s) from a device.

```
>unm^escOUNT (device) p^escRINTER ^cr

>unm ter^escMINAL (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>UNM TERMINAL ^esc BOTH ^cr
>
```

The unmounted file(s) are closed. For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be unmounted; the default is BOTH.

PRIM DEBUGGER

The PRIM debugger is a table-driven, target-machine-independent, interactive program for debugging a PRIM emulator or a target program running on such an emulator. It is tailored to a specific target machine by tables prepared as part of an emulation tool. Basically, it permits a user to set and clear breakpoints and to examine, modify, and monitor target system locations. Target system assembly language and symbolic names are recognized, and arithmetic is performed according to the conventions of the target machine. The debugger command prompt character is "*"; each level of subcommand adds another "*" to the prompt.

ARGUMENTS

Most debugger commands take arguments in the form of values, expressions, expression-ranges, lists of expressions, or lists of expression-ranges as defined below.

Values

A value is an assembly-language instruction, a form, text, or an expression-list. Assembly language instructions are parsed by a table-driven assembler/disassembler that accepts the same syntax as the assembler for the target machine. User symbols will be recognized if they have been supplied in user symbol-table files (see the exec Symbols command) or have been declared individually (see the debugger New-symbol command).

A form requires that the user previously define a corresponding format (see the debugger Format command). A form is represented by the format name followed by an expression-list, as in the following example.

F1 0, 7, 3

Text is represented as a double-quote ("), followed by an arbitrary delimiter character, followed by a sequence of other (non-delimiter) characters, followed by another occurrence of the delimiter character, as in the following example.

"/This is text./

Expressions

An expression is any well-formed sequence of constants and symbols that are defined for the target machine; the symbols (which are machine-specific) may represent either locations or operators whose rules of combination determine what is a well-formed expression. A location symbol may represent a named hardware element or a globally or locally defined user location. An operator may either be unary (preceding its operand) or binary (coming between its operands in infix notation). The precedence of operators is a function of the target machine, except that all unary operators are assumed to have the same precedence value, which is higher (more strongly binding) than that for any binary operator. If brackets are permitted (*e.g.*, parentheses), their precedence value is higher than that of unary operators. For example, A-B and -B+A will evaluate the same, but will differ from -(B+A), which will evaluate the same as -B-A. A bracketed subexpression may itself attain the full complexity of an expression. The behavior of operators is machine-specific.

Expression ranges

An expression-range consists of the triple: expression (lower bound), colon, expression (upper bound). It represents a sequence of locations starting at the lower bound and continuing through successive locations to include the upper bound. The upper bound may not be less than the lower bound. Wherever an expression-range is allowed, a single expression is accepted and treated as if it had been entered as both the lower and upper bounds of a range. If the two bounds in a range address different spaces (see the discussion of Spaces below) within the target machine, the sequence of locations is restricted to that space addressed by the lower bound. Two special forms of expression ranges are recognized. If the second expression in a range is "-*1*", it is treated as being the largest address in the space referenced by the first expression. If the second expression in a range is of the form "*+ expression*", it is treated as if it were "*(lower bound) + expression.*"

Lists of expressions or ranges

A list of expressions consists of at least one expression, followed, optionally, by any number of occurrences of a comma followed by an expression. A list of expression-ranges has the corresponding structure of at least one range, followed, optionally, by any number of occurrences of a comma followed by a range. An example of a list of ranges is

0:10, 20, 30:50

Note that the second element of the list (20) is an example of a range with a defaulted upper bound.

SPACES

Addressable locations in a target system are organized into constructs called spaces. A space consists of a set of addressable locations that is closed under a successor function and its inverse (a predecessor function). For example, main memory constitutes a space, typically starting at location zero and continuing through an arbitrary number of locations. The successor to the last element of a space is the first element in that space; and the predecessor of the first element is the last one. In some cases, machine locations are grouped into a space for convenience, even when the concept of a successor function for elements of that space has no correspondence in the actual target system. Such a space might consist of testable indicators. The machine symbols are identified in the tool-specific user guide.

For purposes of the debugger, every addressable location in a target system is represented by a pair: (*space, element*). When a range is specified, two such pairs *(a,b):(c,d)* are implied. To avoid ambiguities where *a* and *c* differ, the debugger ignores *c* and treats such a range as a sequence of locations, all in space *a*, starting with element *b* and continuing through element *d*.

SYNTACTIC UNITS

The basic syntactic units the debugger deals with are

1. Literals
2. Symbols
3. Punctuation

Literals

Literals are character constants, numeric constants, or single characters that have some encoded meaning (which may be context-dependent). A character constant is supplied to the debugger as a machine-specific character-constant prefix string followed by a string of data characters of arbitrary length, followed by a machine-specific character-constant suffix string of the general form:

prefix-string character-data-string suffix-string.

If the first character of the suffix string is to be included in the data string, it must appear doubled. Character constants are converted to binary (right justified) and are truncated to fit the element in question. As the form of a character constant is machine-specific, it is described in the tool-specific user guide.

A numeric constant is supplied to the debugger as a machine-specific (and optional) radix-prefix string followed by a string of digit characters followed by a machine-specific (and optional) radix-suffix string of the general form:

prefix-string digit-string suffix-string

The prefix and suffix strings establish the radix within which the digit characters are evaluated. The digit characters for any radix ***r*** are the first ***r*** characters of the set {0,...,9,A,...,Z}.

Coded characters have independent meaning only within certain contexts: at appropriate points in the dialogue they designate a particular debugger command, a mode, a breakpoint type, etc.

Symbols

There are five types of symbols: machine symbols that are assigned to hardware elements in the target machine, predefined opcodes for symbolic instructions, user-supplied names of formats, operators for expressions, and user symbols that can be assigned to arbitrary memory locations. Machine symbols are given in the tool-specific user guide; other symbols are assumed to be familiar to the user.

User symbols are either loaded from a file using the exec Symbols command or individually defined using the debugger new-symbol command. The symbols include both global symbols and program-local symbols that belong to specific named programs. The global symbols are available at all times; the program-local ones only when theirs is the open local symbol table.

Punctuation

Punctuation marks are characters with a predefined syntactic (and usually semantic) role. The punctuation characters are the separators (***comma*** and, in format definitions, ***space***), the terminators (***return***, ***escape***, and, in replacement operators, ***back-slash*** and ***up-arrow***), and a semantics-free delimiter (***space***). ***Escape*** is used as a terminator instead of ***return*** to invoke a subcommand or an additional feature of a command (***e.g.***, in Mode or Breakpoint commands described below).

ERROR DETECTION AND EDITING

Debugger commands are examined for errors as they are entered, character by character. As soon as an error has been detected, a bell (beep) is echoed and further input is rejected, except for the generic editing characters ***back-space, retype, backup, delete***, or ***abort***.

COMMANDS

Debugger commands are all single characters; they can be organized into several groups: debugger control, execution control, display, and storage. Each is listed below. Unless otherwise indicated, the command character is the first character of the command name.

Debugger Control

Debugger Control commands provide for user control over several aspects of the behavior of the debugger. They permit the user to execute commands indirectly or conditionally, to return from the debugger to the PRIM exec, and to control the debugger's representation of data. The Debugger Control commands are:

Use. Calls a designated break-time program as if some breakpoint associated with that program had just occurred. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command); the program itself can be seen using the program-edit command.

```
#Use program ?(number of an existing break program)
#Use-program 2cr
```

If the use command is itself in a break-time program, then a go command executed in the called program causes termination of the calling program as well as of the called program.

If. Tests the supplied expression and, if it is true, executes the following subcommand. A true expression is one whose value is *odd*; relational operators yield a value of one when true and zero when false. The tested expression must be terminated by an *escape*.

```
#If ?(expression)
If 3esc <then> ##Type 0cr
 00: 00   #

#If 2esc <then> ##Type 0cr
#
```

Return. Returns control to the PRIM exec; confirmation is required.

```
#Return (to EXEC) cr
>
```

Mode. Interrogates default and current modes and changes modes. A ***question*** after the command character ***M*** will elicit the default and current mode setting; another ***question*** will list all mode settings and associated mode-code-characters.

```
#Mode ?
Current and (Default) mode settings:
  Feedback       Verbose      (Verbose)
  Output         Bits         (Bits)
  Addresses      Symbolic     (Symbolic)
  Line-format    Dense        (Dense)
  Radix          8            (8)
Type ? for more

#Mode ?
Feedback:
  C      Concise
  V      Verbose
Output:
  B      Bits
  F      Formatted (format-name)
  I      Instruction
  N      Numeric
  T      Text

Addresses:
  A      Absolute
  S      Symbolic
Line-format:
  D      Dense
  E      Expanded
Radix:
  Rn     Radix-base n (1 < n < 37 decimal)
#
```

A list of mode settings is expected following the Mode command; if none is supplied, the default settings are reestablished. If the list is terminated by a ***return***, the current modes are changed. If the list is terminated by an ***escape***, a temporary change is made that applies only to the following subcommand, as in the following example.

```
#Mode Instruction esc ##Type 01234cr
 01234: JUMP 0567
#
```

Modes are established for feedback (verbose or concise); output (bits, formatted, instruction, numeric, or text); addresses (absolute or symbolic); output line format (dense or expanded); and output radix (any base from 2 through 36).

The feedback modes control how debugger commands are reflected to the user: ***concise*** suppresses all "noise" feedback (such as command completion); ***verbose*** enables it. The output modes control the general representation of data: ***bits*** treats a datum as an unsigned magnitude; ***formatted*** treats it as a pattern of bits partitioned into contiguous fields according to a designated format (see Format command); ***instruction*** treats it as a machine instruction and disassembles it; ***numeric*** treats it as a signed value, if that is appropriate for the machine; and ***text*** treats it as a representation of a string of characters. The address modes control whether numeric-mode values are to be converted to symbols (if possible): ***absolute*** suppresses the symbol look-up; ***symbolic*** enables it. The line-format modes control the density of displays: ***dense*** suppresses most

debugger-generated line-feeds so as to show more information per line, *expanded* enables them.

When formatted output is selected, the name of the output format must be specified, as in:

#Mode Formatted F1 cr.

Output radix sets the number base for the representation of numeric data (note that numeric input data self-identify the number base). For example,

#Mode Radix 16 cr

causes current output radix to become hexadecimal.

Format. Permits the user to name and define a format as a list of fields, each of which is a designated number of bits wide. The field widths are supplied as a list of numeric constants (separated by commas or spaces).

```
#Format F1esc 2 4 6 8cr
#

#Mode Formatted F1 esc ##Type 0cr
 00: 00,00,00,00   #
```

If the format command is terminated without having defined a format, all defined formats are displayed, as in

```
#Format cr
 F1 2,4,6,8   #
```

Comment. Following an initial semicolon, ignores all subsequent inputs up to and including a line terminator.

```
#; THIS IS A COMMENT--IT DOES NOT GET INTERPRETED.cr
#
```

New-symbol. Adds a list of new user symbols to the (possibly empty) global symbol table. Each new symbol in the list is supplied as a name followed by a *space* or an *escape* followed by an expression giving its location.

```
#New-symbols ?(((new-symbol) <ESC> (expression))-list)
#New-symbols PATCHesc <at> 070000cr
#Type PATCH,PATCH-1,PATCH+1cr
 PATCH: 00     067777: 00     PATCH+01: 00   #
```

Kill-symbol. Removes a list of user symbols from the open local or global symbol table.

```
#Kill-symbols ?(list-of-user-symbols)
#Kill-symbols PATCHcr
#Type 067777:+2cr
 067777: 00     070000: 00     070001: 00   #
```

Open-symbol-table. Opens a local (program-specific) symbol table if one is specified; the currently open local symbol table, if any, is closed in any case. After this command is executed, the available symbols include the global symbols plus the local symbols of the specified program; if no program is specified, only the global symbols are available.

```
#Open-program-symbols ?(program-name) or not <close the open local symbol table>
#Open-program-symbols cr
#
```

Execution Control

Execution control commands provide for user control over execution of the target program. They permit the user to continue execution, transfer to a designated location, set and clear breakpoints or edit break-time programs, and single-step the target program. The execution control commands are

Go. Passes control to the target machine in its current state. If an argument is supplied, its value is first stored into the program counter. The argument can be an arbitrary expression, so long as it evaluates to a legal memory address.

```
#Go (to) ?(expression) or empty
#Go (to) 01000cr
```

Break. Displays or sets breakpoints in the target machine. The two classes of breakpoints are known as event breakpoints and reference breakpoints. There is a fixed set of event breakpoints defined for any given target machine; each describes a type of event whose occurrence causes the emulator to break if the corresponding event breakpoint is set. The set of event breakpoints always includes (1) every instruction-execution (single step), (2) every branch of control, and (3) every memory write; other events are defined for each machine as appropriate. Reference breakpoints cause the emulator to break when a specific type (read, write, and/or execute) of reference to a specific location occurs. Reference breakpoints may always be set on memory locations; other spaces in which reference breakpoints may be set are detailed in the tool-specific user guide. Any number of reference breakpoints may be set at any time.

The break command followed immediately by a *return* causes all existing breakpoints (*i.e.*, those in the breakpoint data base) to be displayed; if a break-time program is associated with a breakpoint, its number is also displayed. Otherwise, a list of either events or ranges (reference locations) for the setting of breakpoints is supplied. If a list of ranges has been entered and terminated with an *escape*, then a list of read, write, or execute reference-break conditions is specified next (as permitted at those locations); the default is all three types. Whenever a breakpoint is set for an event or a location, any earlier breakpoint for that same event or location is superseded.

If the list of events or break types is terminated by an *escape*, as in the second example below, a break-time "program" may be supplied to be executed by the debugger when the break is encountered. The following commands are permitted within such a break program: Clear, Comment, Debreak, Evaluate, Go, If, Jump-history, Locate, Mode, Open, Set, Type, and Use. Replacement within a locate or type command is not permitted in a break-time program. Any number of commands can be included in a break program; the program is terminated by an empty command (terminator only).

```
#Break (at) ?(event-list) or ((expression-range)-list) or <RETURN>
<? for list of events>
#Break (at) 0123:0456, 0712esc (after doing) cr
<R,W,X>    #

#Break (at) 01000esc (after doing) Xecute esc
 ##Type 0cr
 ##Go (to) cr
 ##cr
<Program number is [1]>    #

#Break (at) TICKcr
#
#Break (at) cr
0123-0456 <R,W,X>    0712 <R,W,X>    01000 <X>[1]    TICK <event>    #
```

During program execution, if an event break is detected, or if a reference break (read, write, or execute) is detected at a location for which the corresponding break type has been specified, then execution is terminated before beginning the next target machine cycle and control passes to the debugger to process the break. If a break-time program has been supplied for that break event or location, the program's commands are executed in order by the debugger until either a go command or the end of the program is encountered. If several breaks occur on the same cycle, the program associated with each of them is executed; the order of break-program execution corresponds to the order in which the breaks are reported by the emulator. If every break causes execution of a Go command, then the target program is automatically resumed, provided there is no ambiguity as to where execution is to resume. Otherwise (*i.e.*, if any break had no program or failed to execute a Go command), a message describing each of the breaks is displayed and the normal command level of the debugger is entered.

Debreak. Clears event breakpoints or reference breakpoints at locations in the target machine. The default is to clear all breakpoints. Examples of debreak commands are

```
#Debreak (from) 0234:+4cr
#Break (at) cr
0123-0233 <R,W,X>    0241-0456 <R,W,X>    0712 <R,W,X>    01000 <X>[1]
TICK <event>    #

#Debreak (from) esc all [confirm]cr
#Break (at) cr
#
```

Program-edit. Displays a designated break-time program or permits it to be edited. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command). If the command is terminated by a ***return***, the entire program is displayed; if by an ***escape***, the program is displayed line by line for editing.

```
#Break (at) STEPesc
 ##Type @OLDCCcr
 ##Go (to) cr
 ##cr
<Program number is [2]>   #Break (at) cr
0123-0233 <R,W,X>   0241-0456 <R,W,X>   0712 <R,W,X>   01000 <X>[1]
TICK <event>   STEP <event>[2]
#Program-edit ?(program-number) (<ESC>-to-edit or <RETURN>-to-view)
#Program edit 2cr
   Type @OLDCC
   Go (to)
#
```

When editing a line of a break-time program, the user can specify that the next (\) or prior (↑) line be displayed or that a replacement (R) of the current line or an insertion (I) in front of the current line be made. Editing is terminated by an empty editing specification. Replacement or insertion is identical to the specification of a break-time program within the break command in that a subcommand mode is entered where successive break-time commands can be entered until an empty command is supplied; then editing continues with the next line of the program. An extra (dummy) last line is added when editing a program so that new commands can be inserted at the end; the dummy line is discarded when the command is terminated.

```
#Program-edit 2esc
   Type @OLDCC   :?(↑ <prior>) or (\ <next>) or ((I<nsert>) or (R<eplace>)
 (commands))
   Type @OLDCC   :Replace
 ##Mode Instruction esc ###Type @OLDCCcr
 ##cr
   Go (to)   :cr
#Program-edit 2cr
   Mode Instruction  ###Type @OLDCC
   Go (to)
#
```

Single-step. Transfers control to the target program through the program counter for execution of one instruction. The single coded character ***line feed*** effects this command.

Display

The display commands permit the user to search or examine the contents of designated locations (and, in two cases, optionally permit their replacement) or to evaluate expressions. The commands are:

Type. Displays location and contents of a list of expression-ranges, permitting the contents of each location to be replaced if the list is terminated by an *escape*, as in the following example.

```
#Type ?((expression-range)-list) optional-<escape>-to-modify
#Type 0:2esc    00: 00 = 1cr
 01: 00 = 2cr
 02: 00 = 3cr
#
```

The replacement value can actually be a list of expressions, the values of the expression

in the list going into successive locations starting with the one last displayed. If no new value is supplied before the terminator, the existing value is not modified.

```
#Type 0:2esc    00: 01 = 2esc    01: 02 = esc    02: 03 = 1esc    #
```

In Display-with-replacement only, the coded characters ***back-slash*** and ***up-arrow*** can also serve as terminators and perform special functions: ***back-slash*** causes the next location to be displayed for replacement and ***up-arrow*** causes the prior location to be displayed for replacement; both of these terminator characters permit the user to step beyond the limits of the ranges entered as arguments to the Type command.

```
#Type 010esc    010: 00 = 1↑    07: 00 = 2\    010: 01 = 3\    011: 00 = ↑
 010: 03 = 4↑    07: 02 = 5↑    06: 00 = \    07: 05 = \    010: 04 = \
 011: 00 = 6\    012: 00 = 7cr
#
```

The last location displayed by a type command becomes the "open" location, and the location following the last one displayed or replaced becomes the "next" location (see the next four commands).

<u>Same.</u> Redisplays the "open" location (see the Type command). The single coded character ":" effects this command. The commands Same, Prior, and Next are all shown in the following example.

```
#:  02: 01    #↑  01: 02    #\  02: 01    #\  03: 00    #
```

<u>Prior.</u> Displays the location at one less than the "open" location (see the Type command). The single coded character ***up-arrow*** effects this command. See the examples under Type, Same, and Equals.

<u>Next.</u> Displays the "next" location (See the Type command; the mode in which the open location was last displayed determined how far it was advanced to the "next" locations.) The single coded character ***back-slash*** effects this command. See the examples under Type, Same, and Equals.

<u>Equals.</u> Displays the "open" location (see the Type command) as bits or as a number if the current output mode is already bits. The single coded character "=" effects this command. In the following example format *F2* has been declared consisting of four half-word fields.

```
#Mode Formatted F2cr
#:  010: 00,01,02,03    #=  010: 01    #\  011: 02,03,04,05    #\  013: 06,07,00,01
#↑  012: 04,05,06,07
```

<u>Locate.</u> Finds cells in a list of expression-ranges that contain (or do not contain) a specified value, examining only those bits designated by an optional mask, and displays their locations and contents, permitting each displayed value to be replaced if the list is terminated by an *escape*. The comparison value and mask are expressions terminated by an *escape*; the comparison value defaults to "NON 0" and the mask defaults to all 1's. The search is performed over a list of ranges, as for the Type command.

```
#Locate ?((expression) or NON (expression)) <match value defaults to NON 0>
#Locate NON 0esc (with mask) ?(optional-expression) <mask value>
#Locate NON 0 (with mask) esc<not zero> (in) ?((expression-range)-list)
 optional-<ESC> to-modify
#Locate NON 0 (with mask) <not zero> (in) 0:020cr
 00: 01    01: 02    02: 03    07: 05    010: 04    011: 06    012: 07
```

It is important that the comparison value, the mask, and the data be properly aligned. For example,

#Locate 070^esc (with mask) *070*^esc (in) *0:31*^cr

displays all cells from 0 through 31 whose second octal digit from the right contains all 1's.

When the command is terminated by an *escape* the debugger stops after each display to permit replacement, as for the Type command.

#Locate ^esc <non-zero> (with mask) *07*^esc (in) *0:020*^esc 00: 01 = *3*^cr
012: 07 = ^cr
#

Jump-history. Displays the most recent target-program jumps in the order they occurred. The number of such jumps to display (taken modulo the default value) may be supplied.

#Jump-history ?((expression) or (empty <all>))
#Jump-history *3*^cr
01000--0200(2 times) 0300--0100 #

Evaluate. Prints the value of a single expression. It has no effect on the open location and does not permit replacement.

#New-symbols PATCH^esc <at> *070000*^cr
#Evaluate PATCH^esc = 070000 #

Storage

Storage commands change the contents of designated locations without displaying them and without changing the "open" location. The storage commands are

Clear. Clears the contents of a list of expression-ranges to all zero bits. Clearing an event for which a breakpoint has been established causes the event to be deactivated; it may be reactivated with a Set command. This may be of benefit when a break-time program has been associated with the event as the breakpoint data-base entry for that event is not affected.

#Clear 0.3^esc #

Set. Sets the contents of a list of expression-ranges to the value of an expression or (on default) to all one-bits. If the list is terminated by an *escape*, a single replacement expression is accepted; if it is terminated by a *return*, the default value of all 1's is used. The replacement expression is truncated to fit into the designated locations, if necessary. Setting an event for which a breakpoint has not been established (*i.e.*, for which there is no entry in the breakpoint data base) causes the event to be activated for a single occurrence of that event (with no break program associated), after which the event is automatically cleared.

#Set ?((expression-range)-list)
#Set *0.3*^cr

#Set 0.3^esc = *2*^cr
#

TARGET EXECUTION STATE

Target execution is initiated, or resumed, through explicit commands (exec Go, debugger Go or Single-step). Execution proceeds until a terminating event occurs, causing control to return to the appropriate PRIM command level. When execution terminates, the entire emulated context -- including clocks and outstanding IO operations -- is cleanly frozen until the next time execution is resumed. Except for explicit modifications to the context made by the user at the command level, the termination and subsequent resumption of execution is transparent to the target machine. The terminating events are

The target machine halts normally or is interrupted (by the emulator) due to the occurrence of some anomaly condition. A message to that effect is generated. The anomalies being monitored are listed in the tool-specific user guide.

The user enters an ***abort***. The abort character is echoed and, after execution is stopped, a status message is output indicating the point of interruption.

The emulator detects the occurrence of a break condition established by the user via the debugger breakpoint command. The establishment of breakpoints and the subsequent interruption of execution at the time of their occurrence is the primary program debugging tool in PRIM.

An IO error occurs. A message detailing the particular device involved and the nature of the error is output. IO errors always return control to the exec state; the error messages and their meanings are listed at the end of this section.

When one of these conditions occurs, it is logged and execution continues until the end of the current cycle of the target emulator. It is therefore possible for multiple conditions to result in a single stop. When this is the case, the action and message appropriate to each of the conditions is produced.

When a breakpoint is detected, the debug program, if any, associated with each breakpoint is executed by the debugger before control returns to the command level. Should some break program terminate without a Go -- or should there be some break with no break program -- a message describing the break is output and the command level is entered. Otherwise, execution is automatically resumed; the user receives no indication that a breakpoint occurred unless the break program itself produced output.

TARGET I/O

The target machine that runs in PRIM consists of a processor (CPU) in some particular configuration built by the user to resemble the actual configuration required by his programs. A configuration is built -- before execution is begun -- by installing peripheral devices and establishing values for various machine options (see the exec Install and Set commands). After an emulated device has been installed, and before IO operations can proceed on that device, a (TENEX) file or assignable device must be associated with that emulated device (see the exec Mount command). Subsequent IO operations addressed to that device are then performed on the mounted file.

A mounted file may contain either direct device data (binary) or ASCII text; in the latter case, characters are translated between ASCII and the actual device character set as

they are processed. (If the device character set does not include lower case, input lower case letters are converted to upper case before translation.) When the target device is a record-oriented device (*e.g.*, card reader or punch) and the file is ASCII, then each record operation is performed on a line of the ASCII text file, including truncation and/or blank padding on input.

The mount option ***THIS-TERMINAL*** associates the user terminal (the one being used to communicate with PRIM) with a given device. When the terminal has been mounted on some device, then input from the terminal is switched between PRIM and the target machine every time execution is resumed and terminated. The intervention characters, however, retain their intervention meanings. To allow the full ASCII character set to be input to the target device from the terminal, there is a ***control-shift*** escape character defined during target execution. To help distinguish PRIM output from target output directed to ***THIS-TERMINAL***, all PRIM-generated output is prefixed with the herald "--> " at the beginning of a new line. This applies in particular to both stopping messages and typeout resulting from break-time debugger programs.

I/O ERROR MESSAGES

Various I/O errors may occur. When any one occurs, execution -- including the error-generating operation -- is suspended, and control returns to the PRIM exec. When execution is next resumed, the suspended operation is retried unless it has been explicitly canceled by the user using the exec Cancel command.

"File not mounted."

The indicated device has no file mounted. If a file is mounted before execution is next resumed, the operation will be performed then. (An installed device to which no IO is directed need not have a mounted file in order to run.) The operation may instead be canceled.

This message is also produced when an output operation occurs on a device which has been mounted for input only, and vice versa. Again, a second file must be mounted on the appropriate side of the device in order to proceed normally with the program.

"File not open."

The indicated device has an inaccessible file mounted on it. The device must either be reassigned or unmounted and then mounted. The situation is similar to the case above, except for the possibility of reassigning.

"Improper tape format detected."

TENEX files which are mounted on target magnetic tape devices are encoded in a unique internal format that requires such files to be used only for PRIM magnetic tape devices. The mounted file is inconsistent with that format. The device must be unmounted and replaced with a proper tape file.

"Device not installed."

A device that is referenced by the program is not installed. Should the missing device be required, there is no way to continue this session, since device Installation is no longer allowed. Should the reference be a mistake, execution may be continued down a different path (the operation will be automatically canceled when execution resumes).

"ASCII input character not recognized -- ignored."

The last character read from the ASCII input file on the designated device was not translatable into the character set of the device. The character has been skipped over; resuming execution causes the read operation to continue with the next character in the file. The position of the offending character in the file may be determined via the exec Filestatus command, specifying the indicated device.

Any other error indicates a bug either in the emulator or in PRIM. Such errors should be reported.