WORKING PAPER 165

MINI-ROBOT GROUP USER'S GUIDE

PART 1: The 11/45 SYSTEM

by

Meyer A. Billmers

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

June, 1978

Abstract

This USER'S GUIDE is in two parts. Part 1 describes the facilities of the mini-robot group 11/45 and the software available to persons using those facilities. It is intended for those writing their own programs to be run on the 11/45 system.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

TABLE OF CONTENTS

1.0 Introduction

This document describes the resources available to users of the mini-robot group (micro-automation group) of the Artificial Intelligence Laboratory. It is in two parts: PART 1, THE 11/45 SYSTEM, and PART 2, ACCESS FROM ITS.

PART 1 is intended for the user who will be dealing to a large degree with the PDP11/45 operating system, file structure, and system programs such as EDIT, RUG, and MACRO. The user in question will be writing software to run on the 11/45 system, and may refer to this USER'S GUIDE for the necessary documentation of the 11/45 system and associated facilities. The discussion given in the body of this document will be detailed for the sake of completeness, and should be suitable reading for a new user unfamiliar with the mini-robot system.

PART 2, published separately as working paper 166, documents those mini-robot peripherals for which software has been developed on ITS. This partitioning of function relieves the PHOTOWRITER, PHOTOSCANNER, VIDICON or ARM user, who has applications which the ITS utilities can handle, from having to know any details of the 11/45 system itself. Of course, users of those devices who wish to write their own special purpose programs directly on the 11/45 system may still do so; these users are referred to PART 1.

1.1 General description of the system

The mini-robot system consists of two Digital Equipment Corporation minicomputers, a PDP11/40 and a PDP11/45. The PDP11/45 runs the operating system which supports the system software necessary for most user programming applications. There are essentially only two languages supported on the 11/45, FORTRAN and PDP11 assembler language (MACRO). There is a working version of LISP, but core limitations render it impractical for most non-trivial applications.

The 11/45 has associated with it:

1) Two RK05 moving head disks, capable of storing 1.2 million words each

2) 31K words of core memory

3) A Vidicon TV camera ("computer eye") and digitizer for obtaining visual information about a real world scene

4) A light sensitive P.I.N. diode camera, for highly linear, randomly accessible visual input

5) Two mechanical manipulators ("arms")

6) An x-y table, capable of extremely high precision positioning of objects in a plane

7) A high speed link and a low speed link to ITS, the AI laboratory's time-sharing system

8) A GT40 graphic display terminal, used as the system console and for various display programs

9) A GE1200 Terminet, used primarily as a line printer, and

10) Assorted interfaces and other system peripherals, enumerated in Appendix D

The PDP11/40 exists primarily to support a number of special purpose devices, which may be used off-line in conjunction with a program running on ITS:

1) An Optronics photowriter, used to make high quality photographic transparencies from digital picture data

2) An Optronics photoscanner, which can scan transparencies or opaque prints in black and white or color with high resolution, and return a digitized picture on disk

3) An X-Y tablet, with which the user can enter line traces with a special pen, and have the trace digitized and stored on disk.

4) 24K of core memory

5) Assorted system interfaces

In addition, a small operating system is provided for the user who wishes to program the 11/40 for his own applications.

1.2 Operating system references

The operating system on the 11/45 is a modified version of DOS, the Digital Equipment Corporation's Disk Operating System. The modifications to DOS are described in this paper, but throughout it is assumed that the user is familiar with basic DOS. Before continuing, anyone not familiar with DOS is urged to read the following references. In the following, AL denotes sections of interest only to assembly language programmers and F denotes sections of interest only to FORTRAN programmers. Keep in mind that the mini-robot operating system is based on DOS version 8; any references to BATCH should be ignored (our own version of BATCH is described later). The references given should avoid discussion of DOS features which pertain only to version 9.

AL only: PDP11 Processor handbook (1975 ed.) sections 1.6, 2.2, 2.3, 2.4, chapters 3, 4, 5
AL, F: DOS/Batch Handbook (April '74 ed.) part 1, part 2 chapter 2 sections 2.3.1.2, 2.3.1.3, 2.3.1.4, chapters 3, 5, 6 sections 6.1, 6.2, 6.4.1, chapter 7 sections 7.1, 7.2
part 3 chapter 1 sections 1.1, 1.2, 1.6, 1.7, chapters 2, 3, 4, 6
(AL only) part 6
(F only) part 7
(AL and F) part 9, part 10, part 12, appendices D, E, I, J, K

2.0 The mini-robot operating system

The operating system which runs on the mini-robot 11/45 is a modified form of DOS version 8, as described in the DOS/BATCH Handbook. This chapter describes the significant modifications to that operating system.

2.1 Logging on: LO

Most frequently, the system will be powered on and in an idle state, as indicated by the words "WELCOME TO THE MINI-ROBOT GROUP DOS V08-02" on the system console (if the system is not up, see Appendix B). The user should log onto the system by typing

```
$LO N,M
```

where N,M is your user identification code (UIC) which identifies you to the system and specifies your file area on disk. The system will respond with the date and time. However, if that information had been lost due to a crash there will be a delay while the date and time are fetched from ITS. In the event that ITS is down, a message to that effect will be typed on the console and the user should then enter the date and time himself using the DA and TI commands (see DOS/BATCH Handbook part 3 sections 2.8.4 and 2.8.20).

Next, the system will search your mailbox to see if there are undelivered mail or messages waiting to be read (mail is information directed to a single user, messages are information directed to the entire community). If you wish to read a given item, type a "space"; typing <CR> (carriage return) will cause the remaining items to remain in your mailbox, where they will be waiting when you next log in.

2.2 Logging on: LOG and LOGIN

Alternatively, one can log onto the system with the commands LOG or LOGIN (they are equivalent). These will serve to specify your UIC to the system, but will omit the steps of searching your mailbox and obtaining the date and time from ITS. It may be desirable to have these steps skipped if you are in a debugging mode and are constantly and frequently crashing and bootstrapping the system, so that the seconds spent in searching your mail directory and setting the date and time may be worth saving.

This is strictly a time-saving means of logging in for use in a debugging, crash-prone environment. For normal system use, the standard LO command should be used.

2.3 Conventions for prompt characters

The following conventions are used for prompt characters throughout this document:

$ - top level prompt by the monitor. Indicates that DOS is waiting for a monitor command.

# - file specification prompt. Indicates that the program is waiting for a file specification. File specifications are of the form #<ofield> "<" <ifield>, where the output field and input fields are of the form DEV:FILNAM.EXT[uic] and where DEV may be DK0 or DK1 and <uic> may be omitted if the file is on your directory.

* - command prompt. Indicates that the program is waiting for a command.

2.4 The escape character: ^C vs. ^D

The DOS method for interrupting a program and returning to monitor level is the sequence ^C and KI(LL). For ease of typing and support of other features, this sequence has been replaced by the single character, ^D. All 11/45 software obeys the convention of using ^D as the escape character, and ^C should not be used (except as noted in a later section for a special application). One caution: ^D should never be used to interrupt a program which is running; running programs should not in general be interrupted until they have printed a prompt character (either "#" or "*") and are waiting for user input. In general, it is not safe to interrupt running programs with either ^C or ^D because I/O in progress cannot finish correctly, causing the system to crash and disagreements to arise on disk between a file and its directory entry.

2.5 ":" commands

Normal syntax for DOS commands is a 2 letter command name following the "$" prompt character. Since the most frequently used command by far is RU, it has been replaced by ":". Thus, to run the editor, one now types

```
:EDIT
```

Programs run using ":" instead of RU may have arguments passed to the program on the same line as the call of the program. Thus, to edit the program PROG on device DK1, one may enter

```
:EDIT DK1:PROG
```

instead of waiting for EDIT's initial file specification prompt character, "#", and then entering the name of the file to be edited.

Multiple arguments may be entered in-line, separated by spaces, where each argument will be treated as one line of input to be passed to the program. Thus, for example:

```
$;PIP DK1:FILE1.MAC/DE DK1:FILE2.MAC/DE
```

will run PIP and cause the files FILE1.MAC and FILE2.MAC to be deleted. If the character "^E" appears on the line, it will be translated into the escape character "^D" and the program will return to top level, as in

```
$:PIP DK1:FILE1.*/DE ^E

$
```

Caution is advised when using this feature to pass multiple arguments to a program, however. Output is turned off while there are unfetched arguments left in the string. (This is so that the prompt characters the program uses to ask for the already-supplied arguments will not appear.) An unfortunate side effect of this disabling of output is that any error messages associated with any of the arguments are also lost. Normally, therefore, only one argument should be passed to the program in-line.

2.6 Device names

The DOS system provides for a two or three letter device name for each device known to the system. In general, any device name may be used as an argument to any system program expecting a file specification (although for some applications, only a file-structured device may be permitted). The devices known to the system, and their device names, are given in the following table.

DEVICE	NAME	
Disk 0	DK or DK0	; system disk
Disk 1	DK1	; user disk
GE Terminet (output)	GE	; line printer
GE Terminet (input)	GK	; keyboard
GT40 (input, output)	KB	; system console
Null device	NL	; output only; output ; is discarded

The two disk drives provide for a strict partitioning of disk space into system storage and user storage. The upper disk, DK0, is always for system use and should have the disk cartridge labelled SYSTEM loaded at all times. The user should NEVER write on this disk. Backup copies of the system disk will be kept in case of a crash, but they may not be current. The user should always mount his disk on the lower drive, DK1, and refer to DK1 whenever writing out files on his directory. Unfortunately, it is the nature of the DOS system that the system disk may not be write-protected, so user care and cooperation are needed to prevent the system disk from being overwritten.

3.0 The GT40

The GT40 display processor (and associated PDP11/05 processor) serves as a general purpose system console for the mini-robot system and also as a special purpose graphics terminal. As a system console, it replaces the traditional TTY terminal; the GT40 can display up to 30 lines of ASCII text (up to 68 characters per line) and can buffer hundreds of lines of text in its memory when loaded with the console display program, DISP. Scrolling is available to allow the user to look at text not currently on the screen -- in effect, the user has a window into the text buffer which he can scroll around at will.

In addition, a number of special application display programs are available to display graphs and two-dimension images, or to aid the user in developing his own display programs.

3.1 Loading the GT40

The GT40 is loaded from the 11/45 system, either from top level or by program control using a system call. In order to be loaded, the GT40 must be running one of the following programs:

1) Any of the display programs described in this chapter
2) A user-written graphic program which conforms to the conventions given in Appendix E, or
3) The 11/05 ROM loader.

Ordinarily, (1) or (2) will be true (DISP, the console program, survives the process of powering the system down, so it should not normally need reloading on a day-to-day basis). Should the GT40 crash, condition (3) can be made true by bootstrapping (see Appendix C).

To load the GT40 from top level, type :DISP (to load the general console program) or :GTLOAD (to load any of the special graphics programs described here, or to load a user-written program). The syntax of the call is

```
:GTLOAD PROG
```

where the file extension, if omitted, defaults to .BIN and the UIC defaults to [3,3]. All of the standard display programs are .BIN files on directory [3,3].

GTLOAD may be used to load multiple programs into the GT40; the names of the files should all appear on the command line, separated by commas. The effect of supplying GTLOAD with multiple arguments is to cause each program, in turn, to be loaded into the GT40. If this is to be meaningful, the address space occupied by the various modules should be non-overlapping, and only the last program listed should have a start address specified. After the GT40 is loaded, the last program loaded in will be started at its start address.

The /R switch may optionally be appended to the last argument to GTLOAD. If it appears, that program is assumed to be a .LDA file instead of a .BIN file, and is loaded into the 11/45 and run. In this case, the next to last program listed is assumed to contain the start address.

To load the GT40 from your program, the macro .GTLD should be declared in the .MCALL statement, and should be called as follows:

```
     .GTLD PTR        or         .GTLD #STRING
```

where:

```
PTR:  STRING
STRING:  .ASCII *<ARGS>*<15><12>
```

and where "<ARGS>" is an argument list with the same syntax as for the :GTLOAD call (optionally including the /R switch, as well), such as: <ARGS>= PROG1,PROG2,DK1:PROG3[100,100]. Calling .GTLD in this fashion causes control to pass back to the monitor (if no /R was specified) or to the user program which was the last item in the argument list (if /R was present). If you want control to return to the instruction after the .GTLD call in your program, the following syntax is necessary:

```
     .GTLD PTR,S        or         .GTLD #STRING,S

PTR:  STRING
STRING:  .ASCII *<ARGS>/S*<15><12>
```

While the GT40 is being loaded by the 11/45, it displays the word "LOADING".

Note that when calling .GTLD from a user program, the area in memory from 50000 to 60000 must not be used, as GTLOAD will occupy this area. If .GTLD is to be called as a subroutine (using the /S convention), the area 50000-60000 may be a user buffer whose contents may be clobbered by the call. This area may not contain part of the user stack, however, or any interrupt routines which could occur during .GTLD, as this will cause .GTLD to be clobbered.

3.2 DISP

DISP is a general purpose console program which displays up to 30 lines of text and buffers many additional lines which may be viewed by scrolling a "window" throughout the text buffer. DISP is the program which is normally running in the GT40, and may be reloaded from top level at any time by typing ":DISP".

DISP has three principal modes of operation: console mode, datapoint mode, and real-time edit mode. The normal mode of operation is console mode. Programs which have need for DISP to be in a specific mode can change modes under program control, so that ordinarily the user need not be aware of the multi-mode status of DISP. Some brief familiarity with the various modes is advised, however, since DISP may get out of agreement with a program's intention for it, necessitating user intervention.

3.21 Console mode

Console mode DISP is the normal mode of operation when the GT40 is being used as a system console. In console mode, DISP displays characters received from the 11/45 program, transmits characters typed at the GT40 keyboard, and performs a number of housekeeping chores. Tabs, rubouts, ^U's, carriage returns and line-feeds are interpreted, and scrolling is supported when in SCROLL mode (see below). In addition, DISP keeps a buffer of approximately 12000 characters received from the 11/45. Ordinarily, the last 29 lines received are displayed on the GT40 screen, and a rectangular cursor indicates the position of the buffer's end, which is where the next character received by DISP will be displayed.

Console mode DISP has two sub-modes, which govern the effect of certain control characters on scrolling. In TRANSMIT mode, all characters typed at the keyboard are transmitted to the 11/45 program for interpretation; in SCROLL mode, certain control- characters are interpreted by DISP to allow the user to scroll a window 29 lines high and 68 characters wide through the text buffer. To enter SCROLL mode, type HOME. To enter TRANSMIT mode, type "LOCK-EOL" (hold down LOCK while typing EOL).

When DISP is first loaded, it will be in console mode, and in SCROLL sub-mode.

In SCROLL mode, the following characters have special meaning to DISP:

(up-arrow) -- begin scrolling the text up (or the window down), towards the end of the text buffer. If already scrolling up, (up-arrow) causes the scroll speed to increase. If scrolling down, (up-arrow) stops downward scrolling.

(down-arrow) -- begin scrolling the text down. If scrolling down, increase speed. If scrolling up, stop upward scrolling.

(right-arrow) -- begin scrolling to the right. If already scrolling right, increase speed. If scrolling left, stop leftward scrolling.

(left-arrow) -- begin scrolling to the left. If already scrolling left, increase speed. If scrolling right, stop rightward scrolling.

(HOME) -- causes DISP to jump the display window to the last screenful.

(LOCK-EOS) -- causes DISP to jump the display window to the end of the text buffer (actually, to the last line).

This effectively clears the screen.

Note that the various ARROWS, HOME, and LOCK-EOS are also the control characters ^Z, ^K, ^X, ^H, and ^]. Thus if the 11/45 program expects to receive any of these characters from the user, DISP will have to be in TRANSMIT mode; in SCROLL mode, DISP will interpret these control-characters as scrolling commands.

Note also that attempting to scroll past the beginning of the text buffer (oldest text) will merely cause scrolling to stop. Scrolling may be performed while DISP is receiving characters; DISP normally keeps the last line of text and the cursor in view by scrolling one line of text off the top of the screen each time a new line of text is received (if the screen is full), but this "cursor following" feature may be overridden by scrolling the display window away from the cursor while charaters are being received. DISP will then allow you to look at the screenful selected. Normally, as new characters are received, old characters are discarded so that the text buffer always contains the most recent (buffersize) number of characters received. If the screen window has been scrolled away from the cursor, DISP will give priority to the characters being viewed; characters shown on the screen will not be overwritten, and as new characters are received, DISP will discard the new characters (ringing the console bell once for each character discarded) until the screen window has been moved off the beginning of the buffer. "Cursor following" will resume once the screen window has been scrolled forward so that the cursor and end of buffer are visible, or if you type HOME or LOCK-EOS.

3.22 Datapoint mode

In datapoint mode, DISP effectively converts the GT40 into a datapoint. This means that the scrolling features of display mode are disabled, and the text buffer contains only the text visible on the screen. (In this mode, like a datapoint, DISP has no "memory".) Datapoint mode is entered under control of the ITS program (see Section 4.7); when that program has completed execution, it restores DISP to console mode.

3.23 Real-time edit mode

When DISP enters real-time edit mode, the rectangular cursor disappears and only an EDIT cursor should be present on the screen. In this mode, special real-time edit commands move the edit cursor around, and only this mode allows for the insertion of text into the center of the buffer (at the location of the edit cursor). Real-time edit mode is entered under control of the program EDIT (see Section 4.2), and return of DISP to console mode is ordinarily handled by EDIT as well.

3.24 Switch-selectable features

DISP features may be enabled or disabled by the switches in the 11/05 switch register, as follows:

(switch 15) -- causes lines which are too long for the GT40 screen to be continued on the next line. A continuation character (down-arrow) at the end of the line will indicate such a continuation.

(switch 14) -- cursor blink

(switch 13) -- edit cursor blink

(switch 12) -- noisy mode. Causes the console bell to ring each time a character is received.

(switch 11) -- literal mode. Causes control characters to be displayed, so that tabs, carriage returns, line feeds, etc. will appear on the screen. Useful if a spurious control character has infiltrated your file, and you need to find it.

(switch 10) -- datapoint mode. If the DISP gets out of phase with your program, so that it is in datapoint mode when you're not talking to ITS, for example, toggling switch 10 will return you to console mode.

(switch 7) -- super-literal mode. Causes meta-characters to display (but only if they are received while this switch is up, unlike switch 11, which toggles the display of control characters as the switch is toggled).

Meta-characters are characters preceeded by a ^\, and can cause DISP to take an action (rubout and clear-the-screen are meta-characters; so are the mode changing commands, the blink-the-next-character command, the italics command) or to display a special character, such as a Greek letter or mathematical symbol (the cursor and the edit cursor are both special characters which are represented by meta-characters). Any meta-characters received while switch 7 is up will be displayed as (right-arrow) followed by the non-meta-character analog. DISP's special character set is enumerated in a documentary message which fills most of the text buffer when DISP is first loaded; to see it, type :DISP and then scroll backwards.

One switch which is not part of DISP but which affects the user at type-in time nonetheless is the two-position switch (incorrectly labelled ON) on the back of the GT40 keyboard. In the ON position, this switch causes every character typed at the keyboard to be upper-case; when OFF, upper and lower-case letters may be typed.

3.3 PICT

PICT is a GT40 program (actually two programs) for the display of digitized pictures on the GT40 screen. PICTB6 displays binary pictures, and PICTB7 displays full 256 grey-level images (although the GT40 only displays 8 grey-levels, so that some image quality is lost). In addition to the picture display, which occupies the upper two-thirds of the screen, PICT maintains a text buffer like DISP's, and displays a display window into the text buffer of 9 lines (full width) in the bottom one-third of the screen. The user may scroll the display window using the ARROW keys, HOME, and LOCK-EOS, as in DISP (but left and right scrolling are not implemented) and may enter TRANSMIT and SCROLL modes, as in DISP. This text area facilitates communicating with the 11/45 program which has generated the image, or with RUG in the debugging stages.

The 11/05 switch register is not implemented in PICT except for switches 0-2, which govern the size of the image. With the switch register set to 0 the picture will be at maximum size (as if switches 0-2 were all on); the user may select a smaller size by placing a number less than 7 in these switches.

The two PICT's are loaded by GTLOAD, (see Section 3.1), and data in a user's memory area may be sent to PICT using the system macros .PCTB6 and .PCTB7. The appropriate MCALL must be issued in the declarations area.

.PCTB6 BUFPTR,(THRESH) --
sends a 64 x 64 point picture to the GT40 in binary mode. THRESH is an optional pointer to a threshold value. The default threshold is 0. All points of intensity less than or equal to the threshold will be considered zero in the binary representation. BUFPTR points to the data buffer containing the picture.

.PCTB7 BUFPTR, (BUKPTR) --
sends a 64 x 64 point picture to the GT40 in normal mode. BUKPTR points to a buckets table. The default buckets, if BUKPTR is omitted, are linear. The bucket table, if supplied, should be 8 words long. A point will be sent at an intensity which corresponds to the first bucket that its value is less than or equal to.

3.4 PLOT and PLTPKG

This section describes the programs PLOT and PLTPKG which provide software support for plotting and examining graphs of data on the mini-robot system. PLOT runs in the GT40; PLTPKG is its counterpart in the 11/45.

The user can display up to 4 graphs at one time, can turn any or all of the graphs on and off with the use of the 11/05's bit-switches, and can examine the function values corresponding to any point on the graphs by scrolling a set of cursors which provide data readout on the GT40. In addition, PLOT supports 9 lines of ASCII text at the bottom of the screen, for use in communicating with the program running in the 11/45. Here too, scrolling is possible, allowing the user access to more than 9 lines (though only 9 lines are visible at one time).

PLOT is loaded into the GT40 under program control, and when the user is finished with PLOT, it may be unloaded (by loading the regular display program, DISP) by executing the command :DISP from top monitor level ($ level) anytime PLOT is resident in the GT40. Most programs that run in the 11/45 with PLOT in the GT40 automatically reload DISP when exited (e.g. via a ^D when in PLTPKG).

The 4 graphs PLOT can display are generally referred to as the upper right, upper left, lower right, and lower left graphs. Since there are two sets of coordinate axes, upper/lower refers to the axis. Left/right refers to the identifying ASCII message that appears to the left and right of each axis -- the message on the left labels the left plot, etc.

Each plot may be turned on by raising one of the 4 bit switches, 0,1, 2 or 3 on the GT40 console. When a graph is turned on, its labelling ASCII message will increase in intensity, and the message Y=+0 will appear below the message. This number is the data readout corresponding to the point on the graph indicated by the cursor arrow and blinking spot. There will be one such number for each of the graphs whose bit switches are turned on. The message X=0 which appears at the left center of the screen indicates the X position of all the cursors (0 is far left). The message R=n which appears below the X value reflects the current setting of the programmable resolution (see below). The arrows may be scrolled (that is, the value of X may be changed) by typing the left and right arrow keys on the upper left-most area of the keyboard. Similarly, the text at the bottom of the screen may be scrolled by typing the up and down arrows.

LEFT-ARROW =>	Scroll cursors left. If already scrolling left, increase scrolling speed of cursors. If already scrolling right, stop scrolling.

RIGHT-ARROW	=>	Same as LEFT-ARROW, but to the right.
UP-ARROW	=>	Scroll text at bottom of screen up. If already scrolling up, increase scrolling speed. If already scrolling down, stop scrolling.
DOWN-ARROW	=>	Same as UP-ARROW, but down.
HOME	=>	Enter "SCROLL" mode. "SCROLL" mode is the only mode in which any scrolling may be done. If for any reason, PLOT should get out of "SCROLL" mode and refuse to scroll, type the HOME key.
LOCK-EOS	=>	(hold down LOCK key and type EOS while holding it). Jumps to the end of the ASCII text in the buffer (where the cursor is). Equivalent to Typing UP-ARROW and then waiting a (possibly) long time.
LOCK-EOL	=>	leaves SCROLL mode. Scrolling will be disabled, and the control-letter equivalent of the various arrows will be transmitted to the user program (for example, UP-ARROW is ^Z and DOWN-ARROW is ^K when not in SCROLL mode).

PLOT has room on the GT40 screen for only the first 256 points of each graph. When graphs are sent to PLOT for display, it will automatically display the first 256 points and truncate the rest. The resolution is initially 1, indicating that the graph is being viewed at maximum detail (every point is being displayed). If the user's graph is longer than 256 points, and he wishes to view the rest of it, there are two options. If the cursors are scrolled off the right edge of the graph, more new points will be brought into view. Or, the user can increase the resolution. As an example, if the resolution is 2, the first 512 points of the graph will be visible, though only every other point will be displayed. In this mode, the points that are displayed may be examined directly by scrolling the cursors; once the cursor points to the area of interest, changing the resolution to 1 will cause a section of the graph centered around the cursor's position to be displayed in detail. And of course, the resolution may be increased as much as desired.

In addition to the arrow keys' role in scrolling as mentioned above, the following control keys are implemented in PLTPKG:

^L - redisplay the graphs at the current resolution, centered around the current position of the arrow (that is to say, with the arrow in the center of the screen).

^A - halve the resolution, then redisplay the graphs centered around the current arrow position.

^S - double the resolution, then redisplay the graphs centered around the arrows.

^W - advance the arrows to the next screenful.

^Q - move the arrows back one screenful.

^E - go to the beginning of the graph.

^R - go to the end of the graph.

The user who wishes to use PLOT will surely also need to have access to it from his program. The package of macros and subroutines called PLTPKG makes that access easy. It is assembled together with the user's program, as follows:

```
$:MACRO

#DK1:PROG,PROG<PLTPKG,DK1:PROG
```

and then the macros are available to the user. If the user intends to assemble with DHTIO, the following will suffice:

```
#DK1:PROG,PROG<DHTIO[3,3],PLTPKG[3,3],DK1:PROG
```

Note that the user program should then contain a .END RCOMM statement (see section 4.1).

The plot package will support left/right cursor scrolling, and data readout. It will also automatically reload the display program DISP if a ^D is typed.

A list of these macros, along with sample calls and an explanation of each, follows.

```
        CLRUL

                -- clears the upper left plot.

        CLRUR

                -- clears the upper right plot.

        CLRLL

                -- clears the lower left plot.

        CLRLR

                -- clears the lower right plot.


        CLRALL
                -- clears all plots.
```

The user should clear a plot before sending over new data to that plot, and should clear all the plots with CLRALL before sending over a new set of plots. All plots on the screen at one time should be the same length, so if you are sending over a single new graph, it should be the same length as the others. To change the lengths of the graphs, first CLRALL should be called.

```
        ULMSG #MSG

                -- sends the ASCII text at MSG to PLOT to
                   be used as a label for the upper left plot.
                   The text should be 7 characters or less, and should
```

end in a 0 byte (e.g. use ASCIZ).
Example: ULMSG #MSG
      MSG: .ASCII/Fn. #12/<0>

URMSG #MSG

-- sends out ASCII text to label the upper right plot.

LLMSG #MSG

-- sends out text to label the lower left plot.

LRMSG #MSG

-- sends out text to label the lower right plot.

LDPLOT

-- loads the PLOT program into the GT40. Should be called before any of the other macros in this package. If using DHTIO, you might want to make a subroutine which does a LDPLOT and include the name of that routine in your ICOMND string.

LDDISP

-- unloads PLOT and loads the standard display program, DISP. LDDISP should be called before the user's program returns to monitor; otherwise, system parameters which PLTPKG has saved will not be restored correctly, and the system may crash.

The user who also uses DHTIO should implement an EX command which executes LDDISP and then does a JMP EXIT so that those system parameters which DHTIO saves can be correctly restored (EXIT is a DHTIO label).

RAD8

-- causes the X coordinate, the resolution, and the data readout to appear in octal on the screen.

RAD10

-- causes the X coordinate, the resolution, and the data readout to appear in decimal on the screen. When in decimal, these numbers will have a "." after them.

ARROW XNUM

-- sets the X coordinate of the cursors to be XNUM. This is for use by the user who wants to move the cursor around under program control instead of or in addition to having it moved by the scrolling keys (see LEFT-ARROW and RIGHT-ARROW, above) and PLTPKG's control-keys. User scrolling will still work and will resume moving the cursor from the position last set by an ARROW statement.

Example: ARROW #100. Also, ARROW R2.

ARWPOS N

-- gets the current arrow position, and returns it in N.
Example call: ARWPOS R1

CENTER

-- redisplays the graphs on the screen, centered around the current cursor (arrow) position, at the current resolution. Same as typing ^L.

SETRES R

-- sets the resolution to R. The R= message on the screen will change to reflect the resolution, but the graphs will not be re-displayed. If SETRES is called before doing CLRALL and sending new graphs, the new graphs will appear at the desired resolution, but if existing graphs are not cleared, then CENTER should be called immediately after SETRES, so that the graphs will be re-displayed at the correct new resolution. Sample call: SETRES #4

GETRES R

-- gets the current resolution in R. Sample call: GETRES R0

SETSH S

-- sets the value of an internal variable called SHIFT to S. SHIFT determines how many new points are shifted onto the screen when the arrows are scrolled off either edge. Default setting is 200. Setting this variable to 256 would cause a whole new screenful to appear, with the arrow on the far left (with the default, the arrow will appear indented 56 points). Setting it to 128 will cause the arrow to appear in the middle. Sample call: SETSH #256.

SNDGRF PTR,CTR,C1,C2,SR,INCR

-- sends a graph to PLOT for plotting. PTR should point to the beginning of the data buffer. CTR should contain the number of words in the buffer (or the number of points in the graph -- they are the same). C1 and C2

are codes to determine which of the four plots
this will be:

```
C1 = 0  C2 = 0  => upper left plot
C1 = 0  C2 = 1  => upper right plot
C1 = 1  C2 = 0  => lower left plot
C1 = 1  C2 = 1  => lower right plot
```

These four arguments, PTR, CTR, C1 and C2 must
be present in all calls of SNDGRF. SR and INCR are optional.
If present, SR should be the address of a scaling
routine to expand or contract the range of the data
so that it will fit onto the graphs on the GT40
screen (the range that will fit is +160 to -160).
SR should scale the number in R3, and return.

IMPORTANT NOTE: The graph in question should be cleared by
the user before SNDGRF is called. Use
one of the CLR macros (CLRUL, CLRUR, etc.)
or CLRALL.

An example of a call:

```
SNDGRF #BUFFER,#200,#0,#1,#SCALER
```

where BUFFER is the data, 200 is the number of
points in the graph, the graph is to go in the
upper right, and the data is to be scaled with

```
SCALER:  ASH #-3,R3
         RTS PC
```

that is, the data is to be divided by 8.

If present, argument INCR should be an increment
(in words) to be added while accessing the
data from the graph table. The default value
for INCR is 1, meaning that SNDGRF will plot
every point in BUFFER. If INCR is 2, SNDGRF

will plot every other point (starting with the first one), etc. This is for use in buffers which have more than one function intermixed. For example, if the format of the buffer is

```
BUFFER: X0
        Y0
        X1
        Y1
        X2
```

etc., then to plot the X's, one might call

```
        SNDGRF #BUFFER,#1000.,#0,#0,#SR,#2
```

and to plot the Y'S,

```
        SNDGRF #BUFFER+2,#1000.,#1,#0,#SR,#2
```

Note: SR, if present, must not disturb any other registers besides R3. The call may have more indirection than in the example, e.g. SNDGRF R0,R1,#0,#1 where R0 contains the address of the buffer, and R1 contains the count, etc.

Note also that BUFFER, the data to be graphed, should be one graph point per word.

*** => There are two scaling routines supplied with the package for the user who needs some scaling done to get his data to fit on the screen, and who doesn't want to do it himself. If the data is all positive, use #ASPO (auto-scale, positive only), and if it is signed, use #ASPN (auto-scale, positive/negative) in the SR position of the macro call for SNDGRF. Both will take the data and scale it to the maximum range that will fit on the graph, e.g. if max(data) > 160 it will scale the data down, and if max(data) < 160, it will scale it up, so that you get the greatest possible magnification. The scaling is done to the displayed data only; original data in the buffer is untouched.

Sample call: `SNDGRF #BUFFER,#200,#0,#1,#ASPN`

Using ASPN on positive data will cause it to be scaled to use the upper portion of the graph (above the zero axis), wasting the lower portion. Using ASPO on positive data will cause the data to fill the entire plot, making the zero axis meaningless and moving zero to the bottom of the plot. Using ASPO on signed data will not work.

BGRAPH

-- begin graph mode. Used only in conjunction with SNDPNT (see below).

EGRAPH

-- end graph mode. Used only in conjunction with SNDPNT.

SNDPNT PNT,C1,C2

-- sends one point to PLOT when PLOT is in graph mode. This is for the user who doesn't want to use SNDGRF, because he wishes to send the graph one point at a time as it is computed, rather than after it is all stored in a buffer. PNT is the data item to be plotted, and C1 and C2 have the same meaning as in SNDGRF (but note that now, after the BGRAPH and until the EGRAPH, points may be sent to any of the 4 graphs, in any order desired.)

If the user does use SNDPNT rather than SNDGRF, he will have to inform PLTPKG himself of the size of the graphs and their names, or else the cursor scrolling will not function correctly. The variable GRFCTR should be set to contain the number of graph points, and the table DSPTBL should contain the addresses of the four graphs, in the following order:

```
        DSPTBL: UL      ; UPPER LEFT GRAPH
                LL      ; LOWER LEFT etc.
                UR
                LR
```

Also, INCTBL will have to be filled in with the INCR values for the four graphs (same as the INCR argument to SNDGRF), and SRTBL will have to be filled in with the addresses of the four scaling routines. INCTBL and SRTBL, like DSPTBL, are four-word tables, and the order of the entries is the same as in DSPTBL.

If SRTBL contains the address of the user's scale routines, they will be executed automatically when SNDPNT is called. Thus, there is no need for the user to call them to scale the data. Once again, they should scale the data to the range +160 to -160. Note however that the automatic scaling feature (ASPN and ASPO) which was available in SNDGRF cannot be used here. The user must provide his own scale routines if SNDPNT is used.

When using SNDPNT, it is not necessary to CLR the graph first. CLRALL etc. will reset the pointers to the beginning of the screen, and should be called when a new graph is being sent. But the SNDPNT user can call the EGRAPH macro when only part of the graph has been sent, type some stuff to his program, call BGRAPH, and send the rest of the graphs. The new points will be appended to the end of the existing graphs.

These variables, GRFCTR, DSPTBL, INCTBL, and SRTBL are already present in PLTPKG, so your program need only fill them in. Remember that if you use SNDGRF they will be filled in automatically. This note applies only to users of SNDPNT.

TIMING INFORMATION: SNDPNT takes approximately 2 msecs. per point, BGRAPH takes 1 msec., and EGRAPH takes 2 msecs.

Please note that in order for the cursor scrolling to function correctly, the user program must have control (and should be waiting for a command from the user). If the user is debugging the program and is in RUG (e.g. at a breakpoint, etc.) scrolling will not work (RUG will try to interpret the scroll characters as RUG commands). In order to use the scrolling feature the user must first proceed his program (type altmode-P to RUG). When using DHTIO, there should be a DHTIO prompt character on the screen, indicating that DHTIO is waiting for a command, when you try to scroll.

3.5 Display programming support

The remainder of this chapter describes the facilities available to the user who wishes to write his own display programs for the GT40. Facilities available are GTMAC, a set of macros which implement a GT40 instruction set for the MACRO assembler; GTROS, the GT40 Trivial Operating System, which implements an extension of the GT40 instruction set, and P, a GT40 program which implements the 9 line mini-DISP feature found in PLOT and PICT, while allowing the user to write his own display routines to use the upper two-thirds of the screen.

3.51 GTMAC

The PDP11/05 processor is, with only minor differences, capable of executing the same instruction set as the 11/40. Thus 11/05 programs may be written as if they were to run on the 11/40, assembled in absolute mode (with the Assembler switch .ENABL ABS) and then loaded into the 11/05 with GTLOAD. The instruction set for the GT40 scope processor is, of course, markedly different, and the MACRO-11 assembler does not recognize any standard mnemonics for the GT40 scope instructions.

To enable the user to program easily for the scope processor, the macro file GTMAC.MAC[3,3] has been made available. GTMAC.MAC contains macros which expand into the various scope instructions. These macros are:

SGM MODE,INTEN,K1,K2,K3 -- set graphic mode. MODE may be CHAR (character mode), SVEC (short vector), LVEC (long vector), PNT (point mode), RPNT (relative point), GRFX (graphplot x), or GRFY (graphplot y). It is a required argument; all other arguments to SGM are optional. INTEN sets the intensity level. If present, it must be an integer from 0 to 7. K1,K2, and K3 are keywords. None or all three may appear; note that if INTEN is omitted, the first keyword must still be the third parameter. Their order is arbitrary, and they may be chosen from the following three sets:

> LPI,NOLPI -- enables or disables light pen interrupts.
> BLK,NOBLK -- blinking is on, off.
> SOLID,LDASH,SDASH,DDASH -- line type (solid, long dash, short dash, dotted dash).

LDA K1,K2,K3,K4,K5 -- load status register A. The keywords K1-K5 may be chosen in any order and number from the following sets:

> HALT - stops the scope
> HI,NOHI - interrupt on scope halt is enabled, disabled.
> LPY,NOLPY - the point of light pen interaction is intensified, not intensified.
> ITX,NOITX - characters are italicized, not italicized.
> SYNC - halt scope and restart on next 60 HZ clock pulse.

LDB NUM -- load status register B. NUM is the 6 bit positive graphplot increment.

The following group of macros uses the parameter conventions:

X6,Y6 are signed, 6 bit coordinates
X10,Y10 are signed, 10 bit coordinates
X10A,Y10A are positive 10 bit coordinates
HIDE is an optional keyword that specifies that

the item is not to be intensified.

SVEC X6,Y6 -- short vector mode instruction
LVEC X10,Y10 -- long vector mode instruction
PNT X10A,Y10A -- point mode instruction
RPNT X6,Y6 -- relative point mode instruction
GRFX X10A -- graphplot x instruction
GRFY Y10A -- graphplot y instruction

As in MACRO programs, the .ASCII / text / statement is used to insert characters into a scope program.

To use these macros in your scope program, specify the following command string to the assembler:

```
$RUN MACRO
#DK1:,DK1:<GTMAC.MAC[3,3],DK1:PROG.MAC
```

and your file, PROG.MAC, will be assembled with GTMAC.MAC and all these macros will be defined.

3.52 GTROS

GTROS, the GT40 Trivial Operating System, is available to the user as GTROS.BIN[3,3], and as the following macros in GTMAC:

SJMPR - position independent scope jump

SJSR - jump to scope subroutine.

SJSRR - position independent SJSR.

SRTS - return from scope subroutine.

SINT - interrupt the 11/05. The argument to this macro specifies the address at which the 11/05 is to be started.

SINTR - position independent SINT.

SINTH - interrupt the 11/05, and halt the scope program.

SINTHR - position independent SINTH.

BELL - ring the console bell.

SEXEC - start execution of the scope processor at the specified address.

SREXEC - restart execution of the scope processor at the last point at which it was stopped by a SINTH or SINTHR.

To use the macros in GTMAC which are part of GTROS, the user must first load GTROS into the 11/05. The procedure is as follows:

```
:GTLOAD GTROS,PROG.BIN (or use the .GTLD syntax in your program)
```

This will load GTROS and then your program which makes use of the GTROS macros. GTROS is loaded from 320 to 740.

3.53 P

P is a utility for the programmer who wishes to write his own display routines. P provides such users with an interface to the 11/45 in the form of a mini-DISP with a 9 line display window which may be scrolled through a text buffer, as in DISP. P receives ASCII characters "typed" by the 11/45 program and puts them in its text buffer; it transmits characters which the user has typed at the GT40 keyboard to the 11/45 program; and it displays the user's scope program in the upper two-thirds of the GT40 screen, and allows the user's 11/45 program to communicate with the scope program, to dynamically change the display. P handles the 11/05 interrupts, and drives all the display (the 9 line text window and the user's display) in "SYNC" mode so that brightness is constant as the amount of display changes.

P has a text mode, in which everything received from the 11/45 is treated as ASCII text and placed in the text buffer, and a control mode, in which data coming from the 11/45 is transferred to the user's display program for control of his display graphics.

To use P, it must be assembled as follows:

```
:MACRO DK1:PROG,PROG/NL:TTM<GTMAC[3,3],P[3,3],DK1:PROG
```

and then should be loaded into the GT40 by the system macro .GTLD or from top level by :GTLOAD (see section 3.1).

When done, PROG should be unloaded from the GT40, and DISP reloaded into the GT40 by

```
 .GTLD #STRING,S
```

where

```
STRING:  .ASCIZ/DISP/<15><12>
```

The user's program should not have a start address; instead, the user should define a scope routine to display his picture, and an interrupt routine to receive data from the 11/45 when P is in control mode.

The user's scope routine should begin at location USCOPE, which he should define. The GT40 code should be written in .GTMAC macros, but the P user *is not permitted to use GTROS macros.* The last instruction of the user scope program should be

```
        SJMP PSCOPE             ; JUMP TO P'S SCOPE ROUTINES
```

The user's scope program will begin to display its picture as soon as PROG is loaded into the GT40. The user's 11/05 interrupt routine will not be called until P goes from text mode into control mode. This happens when the 11/45 program sends a ^A (ascii 1) to P. P will discard the ^A and enter control mode, putting the address of the user's interrupt routine into the interrupt vector. The address of the user's interrupt routine should be contained in a symbol called UINTAD, and the status that should be loaded for user interrupts should be contained in UINTST, both of which should be defined in the user's program. Once in control mode, data interrupts coming from the 11/45 will pass directly to the user's interrupt routine. Two subroutines are available in P to help the user get this data:

```
        JSR PC,GET1C
```

will get one ASCII character of data from the 11/45 interface and return it in R0. This is a seven bit binary number. If the 11/45 wishes to transmit data items which are more than 7 bits long, it must do so in 7 bit pieces. GET1C returns immediately, and will return whatever data is in the 11/45 interface hardware buffer. Call it too often, and you'll just get the same data repeatedly. Ordinarily GET1C should be called once for each interrupt.

```
        JSR PC,PUT1C
```

takes the 7 bit number in R0 and sends it to the 11/45. It does not return until transmission of the data across the interface is complete.

There is no special character analogous to ^A which signals the end of control mode. The decision to go from control mode to text mode is made by the user's interrupt routine. To effect the mode change, the user program should call the macro

```
        TXTMD
```

and then return from the interrupt (RTI).

In addition to UINTAD and UINTST, the user must also define the symbols USCPAD and USCPST. These should contain the address and status of the user's interrupt routine which is to receive control when the scope program halts. The scope program halts inside of P once for each display cycle. This is done to ensure synchronization with the clock, and to enable the various 11/05 sub-programs to modify the display code while the GT40 display processor is not running. (It is *very dangerous* in general for the 11/05 to modify display code while the GT40 display processor is executing display code.) Since the user may wish to modify his display code based on data he received in control mode from his 11/45 program, he is given a handle on these scope halt interrupts so that he may make his modifications in this manner also. If you don't wish to do anything with the scope halt interrupts, you can have

```
USCPAD: SCOPE
USCPST: 340

SCOPE:  JMP PSCPAD
```

On the other hand, if the user wishes to change his scope program at scope halt time, he should have the routine at SCOPE (pointed to at USCPAD) to this. Since the changes are known at character interrupt time, the normal programming practice is for the UINTAD routine to buffer the characters received from the 11/45 in a ring buffer, and for the USCPAD routine to empty this buffer, decide what to do to the scope code based on the data received, and make the scope modifications. Note that the USCPST status should be 340 (priority 7, uninterruptable) so that the user will not receive more characters in the ring buffer while emptying the buffer.

The USCPAD routine should not end with an RTI as UINTAD did, but rather with a

```
        JMP PSCPAD              ; JUMP TO P'S SCOPE HALT ROUTINES
```

so that P can restart the display at the proper time and do its own housekeeping chores. Remember that while you are at USCPAD, the screen is dark and the priority is precluding interrupts, so the user should not take excessive time at USCPAD (more than 16 milliseconds repeatedly will cause noticeable flicker).

IMPORTANT NOTE: the user's program must not end with the normal .END statement. A special macro called END has been defined, and should be used as the last statement in the user's program.

A trivial sample program to serve as an example of the various calls and syntax is PTST.MAC[3,3].

4.0 11/45 System programs

This chapter describes the various system programs available to the 11/45 user. Some system programs are part of the standard DOS package and thus are described in the references listed in section 1.2. Those programs which are "home grown" or which are part of the standard DOS package but which have been modified are described here.

4.1 DHTIO

DHTIO is a programming package designed to simplify input/output drudgery and to allow the neophyte system user to be able to write interesting and useful code quickly. DHTIO provides a command interpreter with an argument passing facility and various I/O and system subroutines and macros. The cost of using DHTIO is several thousand words of memory (though the exact cost is variable and memory requirements may be cut if the user is willing to sacrifice certain classes of features) and a loss of flexibility in doing I/O (certain operations are impossible, such as renaming a disk file or random-accessing a disk file). The DHTIO user is spared a large burden of obscure programming needed to deal with the DOS system calls. The new user is therefore advised to strongly consider using DHTIO until his programming effort gets off the ground. If additional capablity is necessary (and this is rare) he can always convert to DOS I/O.

DHTIO is named for its author, Dave Taenzer.

4.11 Using DHTIO

The user of DHTIO must follow several conventions, which will be the subject of this section.

DHTIO is the file DHTIO.MAC[3,3], and should be assembled with the user's program as follows:

```
:MACRO DK1:PROG,/NL:TTM<DK1:SWITCH,DK0:DHTIO.MAC[3,3],DK1:PROG
```

where DK1:PROG is the user's 11/45 program, and where DK1:SWITCH is a DHTIO assembly switch file (see section 4.16).

Two labels must be present in the user's program: HELLO and COMTAB. HELLO is the address of an information message the user wants printed whenever the command interpreter sees "?" on a command line. The message must be ASCIZ (end in a zero byte). Traditionally, the HELLO message lists the commands available to the user, with a brief description of each. COMTAB should consist alternately of a word containing the two ASCII characters of a command name, and a pointer to the subroutine which executes that command. An example is in order:

```
COMTAB: "AB,ABRTN               ; EXECUTES AB ROUTINE
        "CD,CDRTN
        "EF,EFRTN
        0                       ; MARKS END OF TABLE!

HELLO:  .ASCII/Commands available: /<15><12><15><12>
        .ASCII/AB -- does ABRTN/<15><12>
        .ASCII/CD -- does CDRTN/<15><12>
        .ASCII/EF -- does EFRTN/<15><12><15><12>
        .ASCIZ/For more help, see DK1:HELP.INF[201,201]/<15><12><15><12>
```

The user's program will then consist of the three routines ABRTN, CDRTN, and EFRTN. All three should return via the instruction RTS PC. Note that the registers need not be saved, but in turn DHTIO won't save your registers between commands.

Finally, the user program must end with

```
        .END RCOMM
```

When started, this will place control in DHTIO, where a "*" will be printed. Each time the user types a two-letter command at the keyboard, one of the program's subroutines will be executed.

4.12 The command interpreter

DHTIO allows the user to enter numeric arguments to any of the two-letter commands. The user may enter the arguments on the same line as the command, or they may be omitted. In this case, when the subroutine that implements the command calls to DHTIO for its argument, DHTIO will ask the user for the argument, using an ASCII query which the user supplies, and printing the default value of that argument. (The user enters the default by typing <CR> when asked for an argument.) Multiple arguments may be entered separated by spaces or commas. The macro GPARAM is used in the program to call for a set of arguments. An example will illustrate:

```
ABRTN:  GPARAM TABLE            ; EXECUTED WHEN USER TYPES AB

TABLE:  ABMSG1                  ; AB QUERY MESSAGE IF ARG. OMITTED
        10                      ; DEFAULT VALUE FOR FIRST ARG.
        0                       ; MINIMUM PERMISSIBLE VALUE FOR FIRST ARG.
        100                     ; MAX. PERMISSIBLE
        AB1                     ; LOCATION IN WHICH TO STORE ARG. ENTERED
        ABMSG2                  ; TABLE ENTRY FOR SECOND ARGUMENT .
        10
        0
        100
        AB2
        0                       ; ZERO MEANS END OF TABLE, NO MORE ARGS.

ABMSG1: .ASCIZ/Value of AB1 /
ABMSG2: .ASCIZ/Value of AB2 /
```

Then, at run time, the user may type

```
*AB 20,30
```

or he may type

```
*AB
Value of AB1 (default is 10) ? 20
Value of AB2 (default is 10) ? 30

*
```

Note that typing <CR> at the query enters the default value for the argument, and that entering the character "^" on the command line enters all the default values for all arguments, as in

```
*AB ^
```

This causes AB1 and AB2 to receive the value 10. Attempting to enter an argument which is not within the specified minimum and maximum range will elicit an error message from DHTIO.

4.13 I/O routines

DHTIO provides for a number of subroutines and macro calls to facilitate I/O. I/O may be done to the GT40 only, to the GE Terminet only, or to disk only. Alternatively, output may be done to the "current output device", which may be changed among the three devices listed above under program control or by the user at the console. Output done to the "current output device" will be routed to whichever device is currently selected. Following are DHTIO's I/O functions:

TYPEGT ARG -- sends the single ASCII character contained in ARG
    to the GT40.
    Examples: TYPEGT #'A, TYPEGT R0

TYPEGE ARG -- same as TYPEGT, but to the GE terminet

CHAR ARG   -- same as TYPEGT, but to the current output device

PRINT ARGPTR -- prints an entire ASCIZ string on the GT40.

```
    ARGPTR points to the string.
    Example: PRINT ERROR
             ERROR: .ASCIZ/ Error! /<15><12>

OUTPUT ARGPTR -- same as PRINT, but to current output device

OPENO STRING,[DELETE] -- opens a disk file for output.
    STRING must point to an ASCIZ string containing the
    file specification of the file to be opened. [DELETE]
    is an optional argument; if omitted, and an attempt is made
    to open an already existing file, DHTIO will ask you if
    you wish to delete the old file. If [DELETE] is present
    (its value is unimportant) and the file already exists,
    DHTIO will delete the file without asking.
    Example: OPENO FILE1,D
                FILE1: .ASCIZ/DK1:OUT.TXT/<15><12>

DUMP BUFFER,LENGTH -- dumps  LENGTH bytes from memory
    beginning at location BUFFER to the disk file currently
    open for output.
    Example: DUMP BUF1,#1000    ; dump BUF1 to BUF1+777

OPENI [STRING] -- opens a disk file for input. The file
    specification may be pointed to by optional argument
    [STRING]; if omitted, DHTIO will ask the user for the
    name of the file to open.
    Example: OPENI INFILE
               INFILE: .ASCIZ/DK1:IN.TXT/<15><12>

LOAD BUFFER,LENGTH -- loads LENGTH bytes from the disk file
    currently open for input into BUFFER.
    Example: LOAD BUF1,#1000

JSR PC,RLINE -- gets a line of text from the user into the
    DHTIO buffer called TXTBUF.
```

4.14 Miscellaneous routines

PUSH ARG -- pushes ARG onto the stack

POP ARG -- pops the stack into ARG

```
BIN2D #A1,A2 -- converts the value of the number in A2 to a
    five character ASCII string whose characters are the
    decimal digits of A2, and puts the ASCII digits into
    five bytes starting at A1.
    Example: BIN2D #A1,A2
             A1: .ASCII/XXXXX/
             A2: 129.
        causes A1 to become
             A1: .ASCII/00129/
```

BIN2O #A1,A2 -- same as BIN2D, but converts A2 into 6 octal digits

BIN2A #A1,A2 -- does a BIN2D if the current radix is decimal, and does a BIN2O if the current radix is octal.

Note that BIN2D, BIN2O and BIN2A will take negative numbers - in this case, they will put a "-" sign in the byte before A1 (so the user should leave this byte blank if he anticipates negative numbers).

RADJ A1 -- replaces leading zeroes in the 5 byte string beginning at A1 with blanks (right-justifies A1).

LADJ A1 -- replaces leading zeroes in the 5 byte string beginning at A1 with pad characters which the various output routines (such as PRINT or OUPTUT) will ignore (left-justifies A1).

```
JSR PC,YESNO -- waits for the user to type a character. If he
    types "N", YESNO returns to the location after the call;
    if he types "Y", YESNO skips a word.
    Example: PRINT QUERY       ; ASK USER A QUESTION
             JSR PC,YESNO      ; WAIT FOR REPLY
             BR NO
```

```
     YES: ...                  ; HERE IF "Y"
     NO:  ...                  ; HERE IF "N"
```

JSR PC,UCRNT -- done after a JSR PC,RLINE, this converts
    the characters in TXTBUF to upper-case.

JSR PC,RDRTN -- change radix for all numbers to decimal

JSR PC,RORTN -- change radix for all numbers to octal

JSR PC,KBRTN -- make KB the current output device

JSR PC,GERTN -- make GE the current output device

JSR PC,DSRTN -- make DK1 the current output device

JMP EXIT --   exit to monitor

4.15 Special considerations

ICOMND, if defined in the user program, will be an ASCIZ string of commands (either DHTIO commands or user commands from COMTAB) to be executed by DHTIO before printing the first "*". Thus any initialization routines can be automatically executed.

Example: ICOMND: .ASCIZ/AB CD EF/<15><12>

The character ^C, if typed at DHTIO, causes the routine currently being executed to quit, and causes DHTIO to come to command level.

The character "D", if typed at the command interpreter while it is awaiting arguments, causes the command to be aborted along with any arguments already typed.

All commands may be upper-case or lower-case.

In RUG, executing RCOMM1$G has the same effect as typing ^C to DHTIO.

The user should note that many of the routines which are available to the program are also available at command interpreter level as pre-defined commands. One feature (macros) is available at command interpreter level which is not pertinent to the program. The pre-defined DHTIO commands are:

KB	-- change current output device to KB
GE	-- change current output device to GE
DS	-- change current output device to DK1
FF	-- output a form-feed (^L) to current output device
CR	-- output a carriage-return to current ouput device
RD	-- change the radix to decimal
RO	-- change the radix to octal
EX	-- exit to monitor
EM	-- enter a macro into DHTIO's macro buffer. A macro is a string of DHTIO commands.
PM	-- print contents of macro buffer
XM ARG	-- execute the contents of the macro buffer ARG times

4.16 The SWITCH file

As was mentioned in section 4.11, the user assembles his program with DHTIO.MAC[3,3] and a switch file. This switch file is optional; if omitted from the assembly string, the user will receive all the DHTIO features described in this section. If memory space is critical, and if the user can afford to dispense with certain of the DHTIO features, then DHTIO may be selectively pared by defining certain assembly switches in the SWITCH file mentioned earlier. These switches are:

```
ZNODSO=1        ; suppresses the disk output feature
                ; and saves 2440 bytes
ZNODSI=1        ; suppresses disk input (saves 720 bytes)
ZNOHIO=1        ; suppresses I/O printout in "?" message
                ; (saves 554 bytes)
ZNOMAC=1        ; suppresses macro feature (saves 1260 bytes)
```

4.2 EDIT

EDIT is the 11/45 editor which is specially designed to take advantage of the graphic capabilities of the GT40 and of DISP. The full format of the EDIT call is

```
:EDIT DK1:PROG1<DK1:PROG2,PROG3
```

where PROG1 is the file specification of the output file, PROG2 is the file specification of the primary input file, and PROG3 is the file specification of the secondary input file. The file extension in all three defaults to .MAC.

The primary input file will be the file from which EDIT will read the first page for display on the GT40 screen. Most EDIT commands reference this file, but special commands exist to read from the secondary file. All output of edited text goes to the output file (using NL: for the output file is a convenient way to look at a file using EDIT without modifying it). The most common protocol is for the primary input file to be the same as the output file, with no secondary input file. This is the case when a file is being modified. In this case, there is a special syntax; the user merely types

```
:EDIT DK1:PROG
```

Note that if DK1:PROG.MAC doesn't already exist, EDIT will ask if you wish to create a new file by that name. Typing "Y" will create a new file by that name, and give you an empty buffer to fill.

EDIT will read one page from the file being edited and will display the first 29 lines on the screen. When editing is complete (upon execution of the EX command or the EF command, or ^X in real-time edit mode) EDIT will write out the newly edited file as PROG.MAC, and will keep the old file (prior to the editing session) as PROG.BAK. If the user decides to abort an editing session without writing out the file, he may type ^D. This will cause the .MAC and .BAK versions of the file to remain as they were before the editing session started. Because of the seriousness of these consequences, when the user types ^D he will be asked to confirm his decision: typing "Y" exits and loses the editing done so far, and typing any other character leaves you back in EDIT.

In normal editing mode, DISP will display an EDIT cursor somewhere in the text buffer and the regular DISP cursor at the bottom of the screen. The user types one or two letter commands, including any arguments the command requires (arguments are usually strings of characters and may be several lines long). Commands may be separated by a single alt-mode; commands that require no arguments may be terminated by a <CR>; all commands may be terminated with two alt-modes. Separating commands with one alt-mode is a syntactical convenience only. Commands will not be executed until either a <CR> or two alt-modes is typed. At that time, all commands typed in will be executed. Once the pending commands have been terminated and executed, EDIT redisplays the screen and shows you the new state of the text buffer; the text you see is always centered around the current position of the EDIT cursor.

The EDIT command list is as follows:

B - go to beginning of buffer (move edit cursor to beginning)

A - append a page to the buffer (pages are delimited by ^L's)

I - insert a string. String follows the I, ended with alt-mode

G - get contents of save buffer (put at edit cursor location)

S - search. String to search for follows S, ends in alt-mode.
    Example: Ssearch for this string$$.
    Keep going until end of page, then stop. If no argument,
    (e.g. S$$) search for same string as last S.

N - search. If not found in this page, do a P, search
    again, and keep going until end of file is reached.
    If editing a multiple page file, and an S search fails,
    it is useful to be able to immediately do N$$. This
    will search for the same string as the S search, but
    not be limited the the current page.

P - write out this page, read in new page

M - macro feature. This command is of the form
<N>M<ARG> where N is a count and ARG is a
string terminated by two alt-modes. If
ARG is present (it is optional) the string
is placed in the macro buffer. The contents
of the macro buffer is then executed N times
(N may be 0, especially if ARG is non-null).
Example: 0MSstring$-2D$$ places the commands
into the macro buffer to search for "string"
and delete the "ng", but this won't actually
occur until the user types M$$. Each successive
M$$ will do another search and delete another "ng"

X - save ARG number of lines in the save buffer.
ARG is numeric, such as 3X.

Q - repeat last command line

O - alter ARG number of characters. Does <ARG>D$I
Example: 3Onew string is this$$

D - delete ARG characters. Examples: 4D, -3D

L - advance cursor ARG lines. Example: 5L

K - delete (kill) ARG lines. Example: 5K

C - move cursor ARG characters. Examples: 5C, -4C

? - print out this command list

EN - same as N, but for secondary input file

ET - print ARG lines on the GE terminet.

EX - write out this file and exit to monitor.

EX1 - go to beginning of file (this works for multiple page files by writing the rest of the file out onto a temporary file, and then reading in the beginning. If you need to reference something on an earlier page than the current page, or if a search fails because the search falls off the end of the file, it may be desirable to use the sequence EX1$N$$.

EA - append a page from secondary input file to end of buffer (not at the cursor).

EF - do an EX and then start up a batch file (see Section 4.A). The name of the batch file is the argument; file extension defaults to .BAT, entire file name defaults to PROG.BAT where PROG is the program being edited. Examples: EFA.BT1, EFA, EF

Real-time edit mode has an entirely different command set. In real-time edit mode, each command is a single character, always a control character, and commands are executed as soon as they are typed. There is no syntax for most commands, and the user sees the effect of the command on the screen immediately. The user enters real-time edit mode from regular mode by typing ^R; the rectangular cursor will disappear from the bottom of the screen. Characters other than control characters are self-inserting; so for example, typing an "A" causes an "A" to appear at the location of the edit cursor. The real-time edit commands are as follows:

^R - enter real-time edit mode. If in real-time edit mode, display the list of real-time edit commands. (this list will disappear when the next character is typed).

^A - move the edit cursor to beginning of line

^E - move the cursor to the end of the line

^D - delete next character (after the cursor)

RUBOUT - delete one character before the cursor

^O - insert a blank line to the right of the cursor

^W - delete rest of this line (from cursor to the end of the line), but leave the carriage-return

^K - kill a line. Deletes rest of line, including carriage-return

^S - search for string. As you type the string, it will appear at the bottom of the screen. It should be terminated by two alt-modes

^C - same as ^S, but does an N search (across page boundaries)

^Q - quote the next character. Allows for insertion of control characters.

^U - rub out from beginning of line to cursor

^F - move cursor forward one character

^B - move cursor back one character

^L - refresh the screen

^N - move cursor to next line

^P - move cursor to previous line

^T - move cursor forward 8 characters

^^ - move cursor back 8 characters

^V - move cursor forward 29 lines (one screenful)

^Z - move cursor back 29 lines

^Y - get next page (does a P)

^] - move cursor to beginning of page (does a B)

LOCK-EOS - move cursor to end of page

^\ - does an EX1

^X - does an EF (with no argument)

^@ - puts you temporarily in regular edit mode, allowing one line of commands to be typed at the bottom of the screen, then returning you to real-time edit mode automatically. Terminate with two alt-modes

ALTMODE - go to regular edit mode.

The user should note that there are several internal EDIT buffers. The save buffer is filled with text by using the X command, and that text may be deposited in the selected location with the G command. The macro buffer contains EDIT commands to be executed repeatedly; it is filled by M and also executed by M. The search buffer contains the string most recently searched for (by either S or N) and allows that string to be searched for again without being re-typed. The command line buffer holds the last line of commands, and allows it to be executed again (by using the Q command).

The user should also note that the editor creates temporary files on his directory; these are ordinarily deleted by EDIT as well, so the user should not be concerned with them. If EDIT should crash and the system need to be bootstrapped, these files will appear on the user directory; worse, they will be locked, so that EDIT will not be able to delete them, and attempting to EDIT anything will result in a DOS error message. If this happens, they must be unlocked:

```
:PIP DK1:EDITOR.TMP/UN
```

will do it.

4.3 MACRO

MACRO is the DOS assembler. The description of MACRO in the DOS programmer's handbook is accurate, with the following modification:

The user has several levels of listing file he may create. The command string

```
:MACRO DK1:PROG,/NL:TTM/CRF<DK1:PROG
```

will create a full listing file, using 120 columns (due to the /NL:TTM) and a cross-reference file (CRF).

```
:MACRO DK1:PROG,/NL:TTM<DK1:PROG
```

will create the full 120 column listing file with no cross-reference.

```
:MACRO DK1:PROG,/LI:TTM<DK1:PROG
```

will create a full listing file using only 80 columns.

```
:MACRO DK1:PROG,/NL<DK1:PROG
```

will create a mini-listing file which consists only of the symbol table. This mini-listing file is not really suitable for listing, but satisfies RUG's need for a listing file. Any of the longer listing files above will be adequate for RUG also (see section 4.4) but if the user does not use the listing feature and does not want an enormous .LST file on his disk area, using the /NL switch is a convenient way to get by with a short .LST file which is adequate for RUG's symbol definition needs.

Finally, the user can specify

```
:MACRO DK1:PROG,/NL:TTM/CRF<DK1:PROG/L
```

This final /L will cause LINK to be started when the assembly is finished (but only if there are no assembly errors).

4.4 RUG

RUG is a powerful symbolic debugger which replaces DOS's non-symbolic debugger ODT. RUG takes as an argument the name of the user program. It loads the program into core, along with the symbol table for the program, and allows the user to examine the contents of memory locations and to change the contents. The user may reference symbol names that appear in his program, and may have the contents of a location typed out as a number (decimal, octal, signed, or unsigned), as a symbol in the program, or as a PDP11 instruction. He may type in numbers (octal or decimal), symbols, or PDP11 instructions. He may do integer arithmetic in octal or decimal. Finally, the user may set breakpoints and execute his program in a controlled way, examining the contents of selected locations as the program is executed. It may even be single-stepped (executed one instruction at a time).

Calling RUG with the command string

```
:RUG DK1:PROG
```

will cause RUG to take the following actions: 1) A search will be initiated for DK1:PROG.LST; this file must be present, or RUG will give an error message.

2) The symbol table in DK1:PROG.LST will be searched and the symbols found therein will be defined so that they may be referenced by name during the debugging process.

3) RUG will attempt to locate DK1:PROG.MAP. If found, RUG will read the relocation constants for PROG from it, which will be added to all relocatable symbols in PROG's symbol table (those which have an "R" next to them in the symbol table portion of the listing file). If no .MAP file is found, RUG will warn the user of this fact, and then assume a relocation constant of zero. (Recall that the relocation constant applies only to relocatable symbols; if PROG is an .ASECT with no .CSECT, then the relocation constant is never used.) 4) A search will be made for DK1:PROG.LDA, and if it is found it will be loaded into core. If this file does not exist, RUG will print an error message.

The file DK1:PROG.LST must exist, but it need not be a full listing file. See section 4.3 for the use of MACRO to create partial listing files.

If DK1:PROG.MAP does not exist, a relocation constant may still be supplied to RUG by typing

```
:RUG DK1:PROG/RC:42642
```

If DK1:PROG.LDA was obtained by linking together the separately assembled modules DK1:PROG1.OBJ,DK1:PROG2.OBJ, and DK1:PROG3.OBJ, the user may type

```
:RUG DK1:PROG1,PROG2,PROG3
```

and RUG will load DK1:PROG1.LDA, use DK1:PROG1.MAP for the three relocation constants (one for each module), and read in all the symbols from DK1:PROG1.LST, DK1:PROG2.LST, and DK1:PROG3.LST.

When RUG prints its "#", (if the user just typed :RUG), the user may type just a <CR>. This will cause RUG to enter debugging mode without defining any symbols or loading any user programs into core.

RUG will allow the user to define approximately 600 symbols, and then will print the message "TOO MANY SYMBOLS -- THERE WERE N UNDEFINED SYMBOLS". Extra symbols will merely be ignored by RUG.

All symbols beginning with the letter "Z" and all symbols whose value is less than 100 will not be defined by RUG. This allows the user to have "garbage" symbols in his program which will not occupy space in RUG's symbol table.

When RUG has completed the process of defining user symbols and loading the user program, it will enter debugging mode and prompt with a "*". At this point the user may execute any of the RUG commands described below. Debugging mode may be entered at any time by starting the processor at 17000 if there is a copy of RUG in core.

If RUG is used in conjunction with a user program, the program cannot extend below location 50000 in memory.

4.41 Location Opening Commands

The following commands govern the opening and closing of core locations.

foo/	opens location foo and closes currently open location
foo\	same as above, but opens foo in byte mode
/	when typed at an open location, opens the contents of that location and closes the original location.
^	closes the currently open location and opens the previous location
<CR>	closes the currently open location
<LF>	same as <CR>, and additionally opens the next location
[	same as /, but opens the left hand (source) argument of the currently open location or of the last location opened if none is open
]	same as [, but opens the right hand (destination) argument
<	undoes an indirection chain (of [,/, and] commands) and opens the original open location.

When RUG opens a location, it types out the contents of that location and then allows the user to change the contents by typing something to be entered there. Anything the user types at an open location which is not a RUG command and which makes sense to RUG will be stored in the open location when it is next closed. If the location is closed without anything being typed, its contents will be unchanged. Note that only one location may be open at a time.

4.42 Typeout Modes

RUG has a number of typeout modes which govern the way in which the contents of a location being opened are to be interpreted. Preceeding a typeout mode with one alt-mode will set that mode temporarily (so that it will only have effect until the next <CR> is typed), while preceeding it with two altmodes sets the mode permanently. When RUG is started, it is in instruction mode.

The typeout modes are:

I - instruction mode
S - symbolic mode
C - constant (numeric) mode
A - ASCII mode
R - Radix50 mode
. - decimal constant mode (typed again, goes back to octal constant mode)
"-"- signed number mode (negative constants are printed with a "-". (Typed again, goes back to unsigned mode.)
Q - radian mode

4.43 Typein Modes

Regardless of the typeout mode, RUG will always allow instructions, numbers, symbols, or expressions composed of symbols, numbers and "+" and "-" signs to be typed in to any open location. There are a number of special typein modes which are available, however:

& - enters up to three radix50 characters
' - enters one ASCII character
" - enters two ASCII characters
37. - enters the decimal number 37

4.44 Breakpoints

A breakpoint is a tagged instruction in a user program which, when executed, causes control to return to RUG so that the state of the program may be examined. The following RUG commands govern the use of breakpoints:

$B	sets a breakpoint at the current value of "." (the location currently open, or the last location to have been opened)
foo$B	sets a breakpoint at foo
$$B	types out the locations of all current breakpoints
n$D	if n < 10, deletes breakpoint number n. If n > 10, deletes the breakpoint at location n.
$D	deletes all breakpoints
$G	starts execution of the user program at the program's start address
SYM$G	starts execution of the user program at SYM
$P	proceeds with the execution of the user program. This is valid only after at least one breakpoint has been encountered.
n$P	same as $P, but sets a proceed count of n for the breakpoint being proceeded past. This count will be decremented each time the breakpoint is encountered, and RUG will not receive control until the count has gone to zero, so that the breakpoint will effectively be proceeded past n times.
^P	same as 1$P or just $P

$N	single step. Executes one instruction from the user program and then re-enters RUG.
n$N	Executes n instructions in the user program.
^N	Same as 1$N or just $N
$F	Same as $N if the current instruction is not a JSR. If the current instruction (next to be executed) is a JSR, $F executes the entire subroutine and then re-enters RUG.
^F	Same as $F
^G	Interrupts the user program and enters RUG as if a breakpoint had just been encountered

In RUG the user is allowed to define 8 breakpoints. When a breakpoint is encountered RUG will type a breakpoint message of the form:

```
Bn;SYM/ INC R0 ;  R0/ 123
```

where Bn is the breakpoint number, and SYM is the location of the breakpoint. The user should note that the instruction at SYM has not yet been executed. Other messages RUG uses when entered from the user program are:

SS; -- single stepping

SB; -- subroutine single step (from a $F)

IMR; -- illegal memory reference (non-existant memory or attempting to reference an odd address with a word instruction)

IOC; -- illegal operation code (attempting to execute an illegal instruction)

BE; -- bad entry (RUG encountered a BPT instruction or bit 4 of the PSW when no breakpoint or single stepping was expected)

^G; -- control G (the user typed ^G)

Mn; -- monitor point. One of RUG's monitor points has become true (see Section 4.45)

Note that including a BPT in the user program is often a convenient way to assemble in a breakpoint.

4.45 Monitor mode

Monitor mode allows the user to have RUG monitor his program for the presence of certain conditions, and to enter RUG if those conditions become true.

First the user sets monitor points, which are similar to breakpoints, except that the user specifies the location to be monitored, the value to be monitored for, and the condition to monitor for (contents = value, contents > value, etc.)

RUG will then execute the entire program in single step mode, checking the conditions in each monitor point after each instruction. This is a slow process; RUG, when in monitor mode, runs about 80 times slower than the program executing at full speed. Consequently, monitor mode is only useful in selected applications, but can provide a powerful tool for finding obscure bugs.

The following commands are pertinent to monitor mode:

^E - enters monitor mode. When in monitor mode, RUG will prompt the user with ">" instead of "*". This indicates that the program will be single stepped and will run slowly; this is true even if no monitor points are set. Conversely, even if monitor points have been set, if RUG is not in monitor mode then the program will

not be single stepped and the monitor points
will not be checked.

^L - leave monitor mode.

$M or SYM$M - sets a monitor point to monitor location
SYM (if SYM is omitted, monitors ".", the currently
open or most recently opened location). RUG
will ask for the value, the condition, and a
subroutine. The condition may be any of the
branch instructions, less the initial "B",
e.g. GE, LT, HIS, EQ... the subroutine
should be omitted (type <CR>) unless you wish
to use this feature, which is explained below.
REMEMBER: setting a monitor point has no effect
until you place RUG in monitor mode (with ^E).
Once set, monitor points may be left in place,
and RUG may be shifted into and out of monitor
mode without affecting them.

$$M - lists all currently set monitor points

n$K - if n < 10, kills monitor point number n. If n > 10,
kills monitor point at (monitoring) location n.

$K - kills all monitor points (but does not necessarily
leave monitor mode)

.AND - a flag which, when non-zero, causes RUG to be entered
on a monitor point only when all of the monitoring
conditions become true at the same time. If .AND
is set, RUG will cease to test monitoring conditions
as soon as one is false; thus, if the first test
is the least likely to be true, the monitoring
will run much more quickly.

^T - when typed the first time, enters trace mode. This
effectively sets an internal monitor point and
does a ^E. The internal monitor point records

the value of PC each time an instruction is executed, and stores them in a ring buffer of size 26. Successive ^T's print out the buffer contents. Thus, if the user program is jumping into data and halting, trace mode will show you where the program was for the last 26 instructions before it halted.

The user may set up to 8 monitor points.

Routines which run at interrupt level will not be monitored unless bit 4 of the PSW entry of the interrupt vector is set.

If the user specifies a subroutine when setting a monitor point, then the user subroutine will be run once for each instruction executed in the single stepping process, and it will be allowed to decide whether the RUG should be entered. The user subroutine should:

1) not modify any registers

2) set the C bit if it wishes RUG to be entered on this monitor point

Note that the user subroutine may not look directly at the registers, since they will contain RUG's values, not the program's values. However, RUG will pass the subroutine a pointer to the first word of the MPT block; specifying that the monitor point should monitor location R0 when you set it up will cause RUG to give the subroutine a pointer to the user program's registers.

Examples of the use of monitor mode:

EX1: You want to know if R2 is ever > 100.

```
R2$M
VALUE: 100
CONDITION: GT
SUBR.:
```

For the rest of the examples, we will abbreviate the above process as R2 <GT> 100

EX2: You want to know if R0 is ever 0 when in the subroutine FOO. Suppose the next label after subroutine FOO is BAR:. Then three monitor points are needed: R0 <EQ> 0, PC <HIS> FOO, and PC <LO> BAR. In addition, the .AND flag must be set.

EX3: You have a subroutine called FOO, which takes its argument in R0 and returns the value of the function FOO(R0) in R0. You want to monitor for FOO(R0) > 1000. First, set the monitor point

```
R0$M
VALUE: 1000
CONDITION: GT
SUBR.: PATCH
```

and then enter, somewhere in unused memory,

```
PATCH:  MOV R0,-(SP)     ; BECAUSE WE CAN'T MODIFY IT
        MOV @(R0),R0     ; GET THE USER'S R0 FROM RUG-SUPPLIED POINTER
        JSR PC,FOO       ; COMPUTE FOO(R0)
        CMP R0,#1000
        BGT PATCH1
        MOV (SP)+,R0
        CLC              ; THE TEST IS FALSE HERE. DON'T ENTER RUG
        RTS PC

PATCH1: MOV (SP)+,R0
        SEC              ; THE TEST IS TRUE HERE. ENTER RUG.
        RTS PC
```

4.46 Miscellaneous commands

n$T   types out the contents of the next n locations on
       the GE Terminet

```
foo$E  searches for words which reference effective address
         foo (such as the offset word in an index mode
         instruction)

foo$W  searches for words which contain foo

foo:   when typed at an open location, causes the symbol
         foo to be defined and equal to the address of
         the open location

N!foo  defines the symbol foo to be equal to N

foo^K  half-kills foo, so that RUG will suppress its use
         when doing typeout but will still accept it when
         doing typein

symbol$L causes the value of symbol to be monitored constantly
           in the 11/45 DATA lights (when the rotary switch
           is in the DISPLAY REGISTER position).

^D     returns to the monitor
```

The following are some useful RUG locations:

```
 .H   -- highest core address
 .B+n -- location of breakpoint number n (if not set, this
           location will contain the symbol NOBKPT)
 .C+n -- proceed count for breakpoint n, normally set
           by doing m$P
 .RS  -- RUG's program status word
 .PS  -- user program's program status word
 .M   -- mask used for $E and $W searches, to be ANDed with
           items being matched
 .M+1 -- low core limit for searches
 .M+2 -- high core limit for searches
```

Note on searches -- if a $E or $W search succeeds, it will type the address at which it first succeeds. To continue the search, type "!", and to terminate the search, type <CR>.

4.5 PRINT

PRINT is a system program for listing files on the GT40. The format of the call is

```
:PRINT FILESPEC
```

PRINT will print the first screenful, then the word **MORE** and wait for a user reply. If the user types "space", PRINT will list the next screenful; if the user types anything else, PRINT will abort output of that file, print a "#" and wait for a new FILESPEC, unless the user typed ^D (in which case, PRINT returns to the monitor). PRINT can buffer "space"s, and advance the correct number of screenfuls.

4.6 LIST and RELIST

LIST is a system program for listing files on the GE Terminet. The format of the call is

```
:LIST FILESPEC/SW1/SW2/...
```

where the switches SW1, SW2 etc. are optional and may appear in any order, from this list:

/HE -- print a two-page header with the name of the file in BIGPRINT, and the date and time of the listing.

/H1 -- print a one-page header with the name of the file in BIGPRINT

/H2 -- same as H1, but print an extra form-feed first, to reverse the paper folding parity

/NF -- no form-feeds. Suppresses the form-feeds that LIST
    will ordinarily put in every 60 lines

LIST has one very unique property not shared by most system programs: it runs in background mode, while the user is running other programs. After the bigprint header is typed out (if any), LIST returns to monitor. The listing on the GE Terminet continues, and runs even if the user logs out. The only unsafe operation while a listing is in progress is to attempt another listing. The only ways to stop a listing once in progress are to bootstrap the system or to exit from a program such as PIP with a ^C and a .KI (exiting with a ^D will not disturb LIST). If in fact the system does crash and you need to bootstrap while your listing is in mid-stream, it may be resumed where it left off by typing

```
:RELIST
```

4.7 ITS

ITS is a system program which allows the user to log in on the AI ITS timesharing system. The user types:

```
:ITS
```

and the screen clears, DISP switches to datapoint mode, a ^Z is sent to ITS to attract its attention, and the 11/45 user is logged in to ITS using his ITS login name (if the 11/45 system knows who you are). When you are done using ITS, log off there as usual, and type "^^" to terminate :ITS and break the connection to ITS. When exited via "^^", ITS will also switch DISP back to console mode.

ITS will print the message "ITS is down" if ITS does not respond to the initial ^Z in 30 seconds.

4.8 SEND and RECEIV

SEND and RECEIV are programs for transmitting files from the 11/45 system to ITS, and vice/versa. The format of a call is

```
:SEND FILESPEC/SW1/SW2...,FILESPEC/SW1/SW2...
```

The switches, which may appear in any order, are selected from the following list:

/A -- ASCII file (default)

/B -- binary file

/C -- contiguous file (RECEIV only). RECEIV will make the file contiguous on the 11/45.

/P -- picture file (old format)

/TO:USER (SEND only)
/FR:USER (RECEIV only) - sets the ITS directory name to which/from which the file is to be sent/received. SEND/RECEIV know about most 11/45 users, will send/receive files to the user who is logged in on the 11/45. The TO and FR switches are used to override that.

/FI - indicates that this file is not to be sent/received, but rather to be indirected through, e.g. taken as a list of files to send or receive. This list may contain switches or anything else you could type from the keyboard, but it may not contain another /FI.

/GT and /LT - cause the file being sent to ITS to be named on ITS with a file extension of ">" or "<", regardless of the name of the file on the 11/45. Thus, sending

```
#DK1:FOO1.MAC/GT,FOO1.MAC/GT,FOO1.MAC/GT
```

will result in three files appearing on ITS: FOO1 1, FOO1 2, and FOO1 3 (assuming there were no FOO1's already there).

/EX - exits to the monitor after the file it is attached
to has been sent/received. It also logs you off ITS.
Ideal for use in BATCH files (see Section 4.A).

Note that ^D exits to monitor (but *only* type it after the prompt character "#" has appeared) and also logs you off ITS.

Note that SEND allows the use of the wild-card feature "*" in much the same way that PIP does. "*" may appear in place of either the filename or the file extension for any file specification; SEND will find all files with that file name (or with that file extension) and SEND them all.

SEND and RECEIV, like ITS, will time out after 30 seconds if they fail to reach ITS. In this case, the user will get an "ITS is down" message. In times of very heavy ITS use, it is possible to get the "ITS is down" message even if ITS isn't down, if the response time is greater than 30 seconds, or if there are no free job slots. If the user suspects this to be so, he may try typing :ITS to see what ITS's status is, or he may try the :SEND/RECEIV over again.

4.9 USERS

USERS prints out the current list of system users, including the UIC for each user and the ITS login name which ITS, SEND and RECEIV will use when communicating with ITS for that user.

4.A BATCH

BATCH (also B) is a system program for entering a batch stream, which is a string of text to be used instead of keyboard typein whenever typein is called for. If a batch stream is in effect, programs will fetch their input from the batch stream.

The format of the call is:

```
:BATCH FILESPEC
```

where the file extension, if omitted, defaults to .BAT. The file will be read in, and used instead of typein until it is empty.

Batch files allow the user to perform long operations consisting of executing many programs without the user's intervention. Batch files are most commonly used in conjunction with EDIT. When debugging a program, the most common sequence is to EDIT a change, MACRO to assemble the change, LINK, and RUG to load and debug the program. The command strings to MACRO, LINK, and RUG are likely to be the same each time; making a file called PROG.BAT which contains this protocol saves the user from typing it in each time, and waiting for MACRO to finish so that he can type the command string to LINK, etc. A typical batch file might contain:

```
:MACRO DK1:PROG,/NL:TTM/CRF<DK1:PROG
:LINK PROG,PROG<PROG/E
:RUG DK1:PROG
```

Then, after each change is edited, the user can exit from EDIT by using the EF command (or ^X) and the rest of the protocol will happen automatically.

A batch stream may be typed in from the keyboard, by typing:

```
:BATCH

#<CR>
```

and then entering the batch stream manually, terminated by a line containing only a ".".

If this manual entry of the batch stream is used, the last batch stream to have been typed in may be repeated by typing:

```
:BATCH

#<CR>
$$ (alt-mode,alt-mode)
```

Any batch stream which runs MACRO will be terminated if MACRO returns any assembly errors.

Any batch stream may be terminated by typing ^G. The program which is running will continue until it needs more input from the batch stream (remember, it is dangerous to interrupt it before then), but no more input will be fetched. Instead, input will resume coming from the keyboard.

4.B COPY

COPY is a system program which does disk backup. Each user should have two disk packs, labelled #1 and #2. The #2 pack is intended for backup. Periodically, such backup should be done by running COPY. The sequence is as follows:

```
:COPY
(turn DK0 to LOAD, and remove the SYSTEM disk. Place the backup
disk into drive 0)
DIRECTION OF COPY --
   FROM DK1 TO DK0     (you type the 1 and the 0 -- COPY waits
                        for them. They indicate the direction)

TYPE <CR> TO COPY ENTIRE DISK
<CR>
```

and COPY will copy your disk. When finished, it will inform you if there were any errors in the COPY; soft disk errors are recoverable, so that the message may indicate that there were errors but that the disk has been correctly copied, or it may indicate total failure to copy the disk. When COPY is through, you must reload the SYSTEM disk in DK0 and bootstrap the 11/45.

4.C VERIFY

VERIFY is a DOS system program for verification of the file structure on your disk. VERIFY is described in the DOS/BATCH Handbook. This section provides some examples of how and when to call VERIFY.

Occasionally, when the system crashes during disk I/O operations, the directory structure of the disk may become corrupted -- the bit maps, free block queue, and directory entries may cease to agree. When this is suspected (because of inexplicable DOS errors or any other strange behavior of the disk) VERIFY should be run as follows:

```
AS KB:,5
:VERIFY DK 1 N
```

If the disk file structure is good, VERIFY will run to completion without errors and will return to the monitor with no output. There are two basic types of errors: soft errors and hard errors. Soft errors are indicated by the VERIFY error message LOST BLOCK and the block number, or by FREE BLOCK and the block number. They are "soft" in the sense that VERIFY can also fix them with no damage to any files on the disk. *These errors should be corrected immediately.* To correct LOST BLOCK and FREE BLOCK errors, type:

```
AS NL:,5
:VERIFY DK 1 F
```

There will be no output, and you are done. All other errors involve files on disk. The error message will give the name of the file in question; to correct the error, in general, it will be necessary to delete the file. Often the file will be readable, and may be salvaged by SENDing it to ITS, deleting it, then RECEIVing it. Some files/error combinations will not be readable, and the file must be lost.

After correcting errors of this nature by deleting the file(s) in question, VERIFY should be run again as above (with arguments DK 1 N) to ensure that fixing one error has not uncovered another. Often, deleting a bad file will generate soft errors, and VERIFY will need another pass in fix mode (DK 1 F) to correct these.

Very occasionally, a file will appear which will not allow itself to be deleted in PIP. The user should first attempt to unlock it:

```
:PIP FILESPEC/UN
```

If this fails, the program called SUPERD is designed to delete a certain class of disk-corrupted files which are not otherwise deletable. Type:

```
:SUPERD FILESPEC
```

4.D MAIL and MSGS

MAIL is a system program for directing mail to another user of the 11/45 system, to be delivered the next time he logs onto the system (see Section 2.1). MSGS is a system program for directing a message to the entire user community; each user will receive the message at login time. To send a message, type:

```
:MSGS
```

and, when MSGS requests, type the message (it may be several lines long) terminated by a line containing only a ".". To send mail to a particular user, type

```
:MAIL USERNAME
```

and again, when asked, enter the message and a "." line. If you do not know the USERNAME of the person in question, you may type:

```
:MAIL
```

and then

```
"?"<CR>
```

and MAIL will type the list of known users and their USERNAMES.

If you change your mind while entering a message or while sending mail, type ^D and it will be aborted.

4.E LISP

PDP11 LISP provides the user with most of the useful features of a full scale LISP. Atom print names are restricted to being three characters in length, and the print names should be alpha- numeric. This is a shallow binding LISP with saved values of the variables kept on the stack, along with function calls and associated information (such as function return addresses and parameters being passed). There is only one stack. Currently there is no way for a LISP program to do its own file structured I/O.

The command :LISP will cause mini-robot LISP to identify itself and prompt the user with a vertical bar. Expressions may then be evaluated directly at top level, or the user may type a control-G, which causes LISP to print a "#" and wait for a set of file specifications. The general form of the file specifications is:

```
        #DV:OFILE1.EXT[uic],OFILE2.EXT[uic]<IFILE.EXT[uic]
```

where all standard defaults are recoginized (DV defaults to DK0: and [uic] defaults to the current user).

In this notation, OFILE1 is the primary output file, OFILE2 is the secondary output file, and IFILE is the primary input file. If a non-file structured device is specified, no filename, extension or uic need be supplied. The primary input file and one of the output files must be supplied.

Once this file specification has been typed in, followed by <CR>, control will be transferred to LISP. If the primary input file was KB: or GK:, LISP will prompt the input device with a vertical bar and wait for an S-expression. <CR> and <LF> may be used as atom name separators, but only a matching number of right parentheses will close an expression. LISP will then ignore any other characters typed except a following <CR>, evaluate the expression, and send its value to the primary and secondary output files. When the output is complete, LISP will once again prompt the input device with a vertical bar.

If the input file specified was other than GK: or KB:, LISP will read in, evaluate, and output each s-expression on the input file. When the last s-expression has been read and evaluated, LISP will prompt the user with another vertical bar. Bindings established from previous inputs remain in effect.

4.E1 LISP control characters

LISP recognizes the following control characters on an interrupt level, regardless of the device open for input.

CONTROL-D returns control to the monitor.

CONTROL-P suppresses output to the primary output file for the duration of the s-expression currently being EVAL'd. If output is off, CONTROL-P will resume output.

CONTROL-S similar to CONTROL-P, but for the secondary output file.

CONTROL-G terminates computation of LISP, closes all input and output files, prints a "##" and waits for a new set of file specifications. When <CR> is typed, LISP will resume its computation, but it will take input from the new input file.

CONTROL-H is the debugging interrupt. It interrupts LISP but preserves the state of the LISP computation and the state of all I-O. It will temporarily open the GK: or KB: (whichever was open last) for input. LISP will respond by issuing the error message ERROR: EXTERNAL INTERRUPT, and come to top level with a vertical bar.

The user may then examine the state of his program, change bindings, and type (CNT) to close the keyboard for input, and resume computation and I-O from the previously opened files.

4.E2 File Specification Error Messages

LISP maintains the following set of error messages which pertain to the file specification entered at a "#":

SYNTAX ERROR

This error indicates an error in the file specification syntax.

LINKBLOCK ERROR

This error indicates that the monitor did not have sufficient space for a buffer for one of the devices specified.

I-O ERROR
FILE TYPE: PI   ERROR CODE: B

This error indicates a problem with one of the files in the file specification. PI indicates that the file causing the error was the primary input file; PO and SO are the other possibilities. The error code is the ASCII character obtained by adding 100 to the error code in the FILEBLOCK (see DOS monitor manual, pages 3-82, for a complete list of these error codes). The error codes which can appear as a result of an improper file specification to LISP are:

@ - an attempt was made to open a file for output twice.

B - file does not exist; or an attempt was made to open a file for both input and output.

F - file protection violation.

J - the UIC referenced is not known to the system.

L - an attempt was made to create an output file with an illegal name.

O - the file extension was ".CIL".

END OF FILE ENCOUNTERED
BEFORE END OF EXPR

This error indicates that the input file did not contain enough right parentheses.

4.E3 LISP Functions

The following are standard LISP functions:

LM -- lambda

NLM -- nlambda

EVL -- eval

' -- quote

+ -- addition

- -- subtraction

* -- multiplication

/ -- division

GT -- greater than

GE -- greater than or equal to

NOT	boolean not
CAR	car
CDR	cdr
RPA	replace car
RPD	replace cdr
CNS	cons
NUL	the null predicate
ATM	the atom predicate
NUM	the number predicate
LST	the list predicate
OTH	the none of the above predicate
=	the equality predicate
STQ	quoted assignment
SET	assignment
AND	boolean and
OR	boolean inclusive or
CND	cond
LIS	list. Makes a list of its arguments (may take an indefinite number).
PRG	prog. Takes two arguments, a list of local variables and the prog itself.

RET -- returns a value from a prog.

GO -- jumps to a label within a prog.

GBG -- garbage collector. The value returned is the number of free cells.

CL -- characters to list. This will take input from the input device in the form of a character stream, build a list, and return the list as its value.

LC -- list to characters. Takes a list as argument, and outputs the character stream corresponding to that list on the current output devices.

SYS -- regenerates and restructures the system. (SYS) should be called only in dire instances. SYS takes five arguments, which are the size of the hash table, of atom space, of list space, of the "margins", and the stack. The margins are areas not normally available to the system, but are reserved in the event of a list space overflow or a stack overflow. When an overflow occurs, a soft error message will be generated, and the user will have access to the margin space while he tries to free more space. Overflowing the margin causes a hard overflow error, which is not recoverable. Null arguments to SYS leave the corresponding argument position unchanged.

RST -- restore. Flushes the stack and restores top level bindings to all atoms, then returns to top level evaluation.

BRK -- break. Used to set break points in program execution. It prints out an ID which is determined by its first argument. If the first argument to BRK is between one and twenty, it will print out one of the twenty standard system error messages, and if the first argument is outside this range it will print out BREAK: USER DEFINED, followed by the value of the first argument. In any case, BRK will then print out the values of all other arguments passed to it, and then will return to the top level evaluator.

CNT -- continue. Restarts the program from its state at the occurrence of the last BRK. CNT flushes the stack down to the last BRK block, but restores no atom bindings (thus, there should be no open lambdas on the stack), and then continues operation. CNT may be used after a user defined break point, or after a CONTROL-H interrupt. If called with an argument, CNT returns the value of that argument as the value of the last BRK.

TOP -- top level evaluator

BT -- backtrace. This is a diagnostic routine which has three modes: 1) no arguments. BT will print out the name of every function call on the stack. 2) two numeric arguments. BT will print out everything on the stack from the beginning of the (arg1th) function call and continuing for (arg2) function calls. 3) two addresses. BT will dump core between the two addresses, interpreting everything in between (i.e. pointers into list space will be followed, and the corresponding lists will be printed out).

LISP's internal representations for numbers and for addresses are different; numbers are represented internally by odd integers (N is represented by 2*N+1) while addresses are always divisible by four. When communicating with LISP, the user should prefix all addresses (for example, the arguments to BT in dump mode) with @; all other numbers will be assumed by LISP to be numbers. LISP will also use this convention when outputting.

4.F The STAR macro

STAR is a macro which enables the user program (using DOS system I/O calls only; DHTIO users may not use this feature) to interpret the character "*" in a TTY file specification string in the filename or filext positions as a wildcard, which matches all filnames or filexts on the directory in question.

When called, the macro .STAR (it must be defined in the MCALLs) expands into the code for a subroutine called .STAR. The subroutine should be called as follows:

```
MOV #LNKBLK,-(SP)        ; PTR. TO LINK BLOCK
MOV #FILBLK,-(SP)        ; PTR. TO FILE BLOCK
MOV #COUNT,-(SP)         ; MATCH COUNTER
```

```
JSR PC,.STAR
MOV (SP)+,MATCH3        ; .STAR RETURNS 3 RESULTS ON THE STACK
MOV (SP)+,MATCH2
MOV (SP)+,MATCH1
```

.STAR will match the file specified in the linkblock and fileblock against the directory and device specified in the linkblock and filblock. It will match <COUNT> number of times, and return with details of the <COUNT>th match. It will return <MATCH1> and <MATCH2> as the filname (radix 50) of the <COUNT>th match if the "*" was in the filname position; it will return <MATCH3> as the filext (radix 50) if the "*" was in the filext position (it is legal to have two : *.*). All three results will be 0 if there were no match (at least, if there weren't <COUNT> matches. The user should normally call .STAR with COUNT starting at 1, and increment COUNT until .STAR gives the "no match" indication.

5.0 The 11/40

The PDP11/40 in room 903 supports a number of devices on its own unibus, and has a high-speed link to the 11/45 system and to ITS. The PHOTOWRITER and PHOTOSCANNER may be accessed directly from ITS, or they may be programmed from the 11/40 directly. The 11/40 has associated with it a symbolic debugger, a loader, and a trivial operating system which implements teletype I/O and disk I/O to the 11/45 disks.

5.1 Loading the 11/40

The 11/40 may be loaded anytime it is running its ROM loader. For instructions on bootstrapping the 11/40 ROM loader, see Appendix A.

The program to be loaded into the 11/40 must be assembled with MACRO with the following assembler statements:

```
        .ENABL ABS
        .ASECT

        .=1000          ; LEAVE ROOM FOR TROS

or      .=40000         ; LEAVE ROOM FOR RUG40
```

The ENABL ABS statement will cause MACRO to produce a .BIN file instead of a .OBJ file. This .BIN file need not be LINKed. To load the 11/40, the command syntax is

```
:11LOAD FILESPEC1,FILESPEC2,etc.
```

where the files FILESPEC1, FILESPEC2, etc. are loaded into the 11/40 sequentially. The syntax of 11LOAD is virtually identical to GTLOAD (see Section 3.1). The last module loaded in should have a start address. The last entry on the command line to 11LOAD may have the /R switch; if so, this file is assumed to be a .LDA file and is loaded and run on the 11/45, and the next-to-last file should have the 11/40's start address.

SPECIAL NOTE: There is an unfortunate situation in the 11/40 bootstrap ROM. Due to the limitations on the length of the ROM program, the ROM routine cannot clear location 112, yet it must be 0 before the 11/40 can be loaded. Thus, if for any reason, location 112 becomes non-zero, bootstrapping will not clear it and the 11/40 cannot be loaded. This is normally not the case; but a user program may clobber this location. If the 11/40 doesn't load, check location 112. If it is non-zero, clear it by hand.

5.2 RUG40

RUG40 is a symbolic debugger for the 11/40. Its features and syntax are identical to those of RUG (see Section 4.4). To load a user program and RUG40 into the 11/40, type:

```
:11LOAD RUG40,DK1:PROG.LST,PROG
```

The program must not extend below 40000 to be usable with RUG40. When RUG40 is ready, it will print a "*" on the teletype (which must be turned ON-LINE).

5.3 TROS and BOS

TROS is the Trivial Operating System for the 11/40, and BOS is its 11/45 counterpart, the Background Operating System. Together they implement a set of system macros which handle teletype I/O and disk I/O to the 11/45 disks. Disk I/O is done at background level on the 11/45 while the 11/45 is running another program. The 11/40 user can read or write a disk file even while the 11/45 user is accessing the disk.

To call TROS alone (no disk I/O features), type:

```
:11LOAD TROS,DK1:PROG
```

or, if RUG40 is being used to debug PROG,

```
:11LOAD TROS,RUG40,DK1:PROG.LST,PROG
```

To call TROS and BOS together (TTY I/O and disk I/O):

```
:11LOAD TROS,DK1:PROG,BOS/R
```

or with RUG40 for debugging:

```
:11LOAD TROS,RUG40,DK1:PROG.LST,PROG,DK0:BOS/R
```

All macros which follow must be declared in the .MCALL statement.

Format for .TTYWT (teletype output) is .TTYWT PTR, where PTR points to an ASCIZ string to be printed on the teletype.

Example:

```
        .TTYWT PTR      ; TTY OUTPUT

PTR:     STRING
STRING: .ASCII/THIS IS THE TTY OUTPUT/<15><12>
        .ASCIZ/IT MUST BE ASCIZ TO WORK./<15><12>
        .EVEN
```

Format for .TTYRD (teletype input) is .TTYRD PTR, where PTR points to a buffer into which the line of text read from the TTY will be placed. Unlike .TTYWT,.TTYRD will do only one line at a time.

Example:

```
        .TTYRD #BUFFER
```

Format for .DKOPN (open a disk file) is .DKOPN PTR, where PTR points to a disk name block containing the name, device, and UIC of the file to be opened for input or output. Only one file may be open at a time.

Example:

```
        .DKOPN #DSKNB
```

```
; THIS CALL OPENS DK1:FILNAM.EXT[101,102]

DSKNB:  .RAD50 /FILNAM/
        .RAD50 /EXT/
        .BYTE 101,102           ; UIC
        .BYTE 1                 ; DRIVE NO. DK1. (ZERO HERE FOR DK0)
        .BYTE 0                 ; ERROR FLAG. SET BY .DKOPN

        .WORD ERRRET            ; LOCATION OF THE ERROR RETURN
```

Note that the drive number must be 0 or 1 corresponding to DK0 or DK1. The UIC must be present -- there is no default UIC. In the event of an error, control will be transferred to the error routine (here, ERRRET). The error flag will indicate the type of error. There are two possibilities: error flag = 0 means the file was not found, and error flag = 1 means the file was a contiguous file. TROS will only handle linked files.

The format for .DKRD (disk read) is .DKRD PTR, where PTR points to a header block which contains details of the transfer. Any number of bytes of data may be transferred, but they will begin where the last .DKRD left off. Attempting to read past the end-of-file will give a done indication. If it is desired to read from the beginning of an already open file, it must be closed and then re-opened to position the read pointer at the beginning.

Example:

```
        .DKRD #DSKHB

DSKHB:  DBC                 ; DESIRED BYTE COUNT
        ABC                 ; ACTUAL BYTE COUNT (WRITTEN BY TROS)
        DF                  ; DONE FLAG (END-OF-FILE) (WRITTEN BY TROS)
        BUFPTR              ; POINTER TO THE BUFFER TO BE FILLED.

BUFPTR: .BLKB DBC
```

The desired byte count indicates the number of bytes of data the user wishes read. In general, the actual byte count will equal the desired byte count unless the end-of-file was encountered. If DF is set after the call, ABC tells you how many bytes in the buffer are meaningful.

The format for .DKWT (disk write) is the same as for .DKRD, except that ABC and DF are now meaningless. Note that the same .DKOPN opens an input file or an output file. TROS will not permit two files to be open at the same time, so the user must do all input at once and close the file before doing output, and vice/versa.

.DKCLS closes the disk file currently open.

The format for .CSI (command string interpreter) is .CSI PTR, where PTR is a pointer to the CSI block, as in this example:

Example:

```
        .CSI #CSIBLK

CSIBLK: TTYBUF
        DSKNB
```

and where TTYBUF is the buffer into which a line of text containing the file specification of the file to be opened has been read from the teletype, and where DSKNB is an empty name block, as in .DKOPN.

Example:

```
        .TTYWT #MSG1            ; ASK USER FOR NAME OF FILE
        .TTYRD #TTYBUF          ; GET IT FROM HIM
        .CSI #CSIBLK            ; CONVERT TO A NAME BLOCK
        .DKOPN #DSKNB           ; OPEN THE FILE
        .DKRD #DSKHB            ; AND READ IN A BUNCH OF STUFF!

MSG1:   .ASCII/ENTER FILE NAME/<15><12><15><12>
        .ASCIZ/#/
        .EVEN
TTYBUF: .BLKB 100
CSIBLK: TTYBUF
        DSKNB
DSKNB:  .BLKW 5
```

```
        ERRRET
DSKHB:  1000
        0
        0
        BUFFER

BUFFER: .BLKB 1000
```

6.0 The Devices

The chapter describes the system macros available to interface the user's program to the various devices on the 11/45 and 11/40 unibusses.

6.1 The ANALOGIC converter

The ANALOGIC converter is a black box which implements 8 D/A channels and 64 A/D channels for conversion of data to drive various devices. The D/A and A/D channels are accessible to the user via a patch panel to the left of the ANALOGIC converter itself.

Two system macros exist for use with the Analogic converter. These are .A2D and .D2A. To define these macros in your program, the assembler directive is

```
 .MCALL .D2A,.A2D
```

Note that there are 24 additional D/A channels which are separate from the ANALOGIC converter. The .D2A macro will work for channels 10 to 30 as well. If you intend to use only D/A channels 0-7, the macro .D2AF is faster and shorter; the syntax is the same.

These macros will now be defined, and when called in a user program they will expand into code necessary to do digital-to-analog and analog-to-digital conversions. .D2A takes two arguements:

```
 .D2A CHNUM,VALUE
```

where CHNUM is some variable which contains the channel number (from 0 to 7), and VALUE is a variable which contains the number to be converted. The contents of VALUE may range from 3777 (which will produce nearly +10VDC on the desired channel) to -4000 (-10VDC). This macro requires approximately 5 usec. delay time for the conversion.

 .A2D takes two arguments:

```
 .A2D CHNUM,VALUE
```

It will convert the voltage on the channel contained in variable CHNUM, and will do a busy wait until the conversion is complete (this takes about 10 usec.) The converted value is left in the variable VALUE.

6.2 The X-Y Table

The assembler directive

```
 .MCALL .TABLE
```

will define a macro called .TABLE which, when called, expands into a set of subroutines for moving the x-y table. These routines are called using the convention JSR PC,SUBR. The table subroutines are:

CALTBL	calibrates the x-y table and leaves it in position (0,0)
VELTBL	sets up the velocity for the next table movement. R0 should contain the velocity for x and R1 should contain the velocity for y
ABSTBL	moves the table to the absolute location (x,y), where x is contained in R0 and y is contained in R1
RELTBL	causes the table to move relative to its current location. The x and y in R0 and R1 respectively are taken as offsets for the relative motion and are preserved so that successive calls of RELTBL will reference them.

Note: neither ABSTBL nor RELTBL wait for the table to finish moving. Neither should be called if there is a chance that the table is in motion without first calling WTTBL.

WTTBL	waits for the table's motion to finish. WTTBL will

take a skip return if the table motion completes normally (without running into a limit stop). If a limit stop is encountered, WTTBL will take a non-skip return. Thus:

```
JSR PC,WTTBL
(error return)
(normal return)
```

WHRTBL	returns the table's x position in R0 and its y position in R1.

NOTE: These macros will protect the user from moving the table to a negative position; motion will stop at zero, and the table will not have been decalibrated. Similarly, attempting to move the table too far forward in either x or y will result in a cessation of the table's motion without running into the physical limit stops or decalibrating the table. WTTBL will take the error return whenever such a premature stoppage occurs. Note also that all coordinates kept by the .TABLE routines are relative to the calibration point, and thus CALTBL always should be the first routine called.

6.3 The Photowriter

The photowriter is ordinarily accessed from ITS via PW (see PART 2 of this USER's GUIDE, W.P. 166). The user who wishes to write his own software for the photowriter may use the package called PW1. PW1 is assembled with the user's program, and implements a number of macros which allow for line-at-a-time generation of the picture to be photowritten.

Since the photowriter is on the 11/40 bus, the user program will have to have the MACRO directive .ENABL ABS in order to be suitable for loading into the 11/40 with 11LOAD.

The user program should not contain a .END statement.

The user program should be assembled with PW1, as follows:

```
:MACRO DK1:FOO,FOO<DK1:FOO,PW1[3,3]
```

This will create the listing file FOO.LST, and also the 11/40-executable binary file FOO.BIN. To load this binary module into the 11/40 (after bootstrapping the 11/40 (see Appendix A) the user should execute:

```
:11LOAD FOO
```

or, for debugging sessions,

```
:11LOAD RUG40,FOO.LST,FOO
```

The user program should contain the following .MCALL's:

```
 .MCALL .PW1,.PICT1,.PICT2,.LINE,.POINT
 .PW1
```

(Note that the call of .PW1 must come before any user code.)

There should be three labels inside the user program, called PICTUR, LINE, and POINT. These correspond to the three user subroutines, the three entry points into the user program from PW1. PW1 will transfer control to the user program at the lable PICTUR once for each picture. The address LINE will receive control at the beginning of each new line of the picture, and POINT will receive control once for each point in the picture.

The first thing at the label PICTUR in the user program should be the call of the macro .PICT1, as follows:

```
PICTUR:  .PICT1 RASTER, XPOS, YPOS
```

where RASTER, XPOS and YPOS are locations or registers in the user program. The raster setting of the three-position raster switch on the photowriter will be put in RASTER, and will be either 25, 50, or 100. XPOS and YPOS will be written into with the current location of the photowriter lens, in cms.*100. (e.g. if the lens is at 3,4 then XPOS will be 300. and YPOS will be 400). All three arguments to .PICT1 are optional, and if omitted the values of these variables will be thrown away.

The last item at PICTUR should be a call to the macro .PICT2 of the form:

```
 .PICT2 XSIZ, YSIZ, XPOS, YPOS, RES, NEGPOS, DEMIN, DEMAX
```

where:

XSIZ and YSIZ are the dimensions of the picture, in points.

XPOS and YPOS are the x and y positions on the film at which the picture is to start, in cms.*100. (e.g. XPOS=300.,YPOS=400. for (3,4))

RES is the programmable resolution (see ER command of PW)

NEGPOS is 0 for negative, 1 for positive (see NI, PI commands of PW)

DEMIN and DEMAX are minimum and maximum values for data enhancement (see ED command of PW).

XSIZ, YSIZ, XPOS and YPOS are mandatory arguments. The rest may be omitted and reasonable defaults will be put in (e.g. resolution 1, negative image, no data enhancement).

Remember, too, that arguments like XSIZ may be variables which hold the size, or they may be numbers. If numbers, don't forget the "#".

IMPORTANT: The call of .PICT2 will return control to PW1

At LINE there should be a call of .LINE, as follows:

```
LINE:  .LINE #PTR
```

```
or just:  .LINE
```

If called with the argument, PTR will be taken to be a pointer to a user buffer containing an entire line of points. In this case, the user label POINT will never receive control (it still must be defined, though.)

If called without the argument, it is assumed that the user program will generate the points one at a time, an PW1 will call the user program at POINT to get the points. The last statement in POINT should be a .POINT, as:

```
POINT:  .POINT INTEN
```

where INTEN contains the single byte intensity for the next point. There may be code after the label POINT and before the .POINT which calculates the intensity... in any case, the .POINT will return control to PW1.

Note that when debugging with RUG40, the teletype becomes the system console.

6.4 The Scheinman Arm

The Scheinman arm is normally run from the ITS program ARM, which is described in PART 2 of this USER's GUIDE (W.P. 166). The 11/45 user who wishes to write assembly language programs to move the arm using this controller may do so. The file A implements a number of macros and subroutines which make the same functions available to the ITS LISP user and to the 11/45 user as well.

To use this program, the user should assemble his program as a .CSECT, then link with ARM.OBJ[3,3]. The following system macro definitions should appear in the user program:

```
 .MCALL .ARM,.GO,.GOTO,.STATS,.WHERE,.GTASV,.STOP,.MSTOP
 .ARM
```

Note that the call of the macro .ARM defines the necessary globals for LINK.

ASV denotes an actual state vector; DSV denotes a desired state vector. The subroutines available (in parentheses), and their macro calls, are:

.GO #DSVPTR (GO) - moves the arm to the state specified by

```
                          DSVPTR. Return is immediate. If no
                          other motion is currently happening,
                          GO will start up this request; otherwise,
                          it will queue the request, and take it
                          in turn.

.GOTO #GOPTR  (GOTO) -    moves the arm to the XYZ position and
                          hand orientation specified in the table
                          at GOPTR. The format for this table is
                          given below. .GOTO does a .GO, so
                          return and queuing are the same as for .GO

.STATS #STPTR  (STATUS) - gets the arm status. This will be the
                          address of the DSV currently being serviced,
                          if the arm is in motion; it will be a 0
                          if all movements are done. If the movement
                          which is underway is part of a .GOTO, the
                          the DSVPTR will be an internal one, but
                          STATUS will still return 0 when done.
                          The value is returned in STPTR.

.WHERE #WHPTR  (GETXYZ) - gets the arm position in the coordinate
                          system of the table. The format of the table
                          pointed to by WHPTR is given below.

.GTASV #ASVPTR  (GETASV) - returns the ASV in the 15 word buffer
                          pointed to by ASVPTR. .GTASV is the analogue
                          of .WHEREZ in the frame of reference of the
                          arm's joint angles.

.STOP      (STOP) -       stops the DSV currently in progress. It will
                          servo to the current position vector,
                          with all velocities 0, then set brakes.
                          It does not return until brakes are set.

.MSTOP   (EMSTOP) -       emergency stop. Zeroes all currents and sets
                          all brakes, and returns.
```

The format for a desired state vector (DSV):

```
DSV:    DP0             ; desired position of joint 0. If = -10000,
                        ; then this joint is not to be servoed.
        DV0             ; desired velocity of joint 0. If -10000,
                        ; the user wishes the system to go to position
                        ; DP0 with final velocity 0, and does not
                        ; care how the system gets there. The
                        ; system will choose a maximum velocity for
                        ; this joint which is reasonable for the
                        ; rest of the movement.
        EP0             ; maximum error in position 0 which the user
                        ; will accept. This number is used to
                        ; determine when to set brakes; see note below.
        EV0             ; maximum error in velocity which the user
                        ; will tolerate. See note below for implications
                        ; with regard to setting brakes.

                        ; if EV0 = -10000, EV0 will be assumed to
                        ; be infinity, and when the joint is within
                        ; distance EP0 of the goal, the DSV will
                        ; be finished.
        DP1
        DV1
        EP1
        EV1

        .
        .               ; same as above, but for joints 1-6
        .

        DP6
        DV6
        EP6
        EV6

        LINK            ; if LINK=0, there are no more DSV's
                        ; in this chain. Otherwise, LINK points
                        ; to the next DSV to be done.
```

Note on setting brakes: brakes will be set on a particular joint only if all the following conditions are true simultaneously:

1) The DSV is done, e.g. all positions and velocities are within specified tolerance limits

2) There are no more DSV's, either in the chain of link pointers, or in the queue

3) DVn is 0 for the joint in question, and

4) DPn is not -10000 for the joint in question (code DP=-10000 causes the joint to be skipped over.

Thus, if a joint prematurely finishes and is within its specified error tolerance, it is carried along until all joints are finished before its brake is set. Conversely, a joint which is still in motion when the DSV is finished will not have its brake set.

Format for an actual state vector (ASV):

```
ASV:    POS0            ; actual position of joint 0
        VEL0            ; actual velocity of joint 0
        POS1
        VEL1

        .
        .
        .

        POS6
        VEL6
        BRAKES          ; contains a bit =1 for each brake that
                        ; is set (bit 0 <==> joint 0, etc.)
```

Format for a GO table (argument to .GOTO):

```
        XPOS            ; X position of hand
        YPOS            ; Y position of hand
        ZPOS            ; Z position of hand
        ALPHA
        BETA
        GAMMA
        CONFIG          ; Euler angles for the orientation
                        ; of the hand, and the configuration
                        ; of the arm (see W.P. 69).
        VEL             ; velocity. Maximum velocity to be
                        ; used by any of the joints in attaining
                        ; this configuration. Range is 10 to 300.
```

Format for the table returned by .WHERE:

```
        XPOS            ; X position of hand
        YPOS            ; Y position of hand
        ZPOS            ; Z position of hand
        ALPHA
        BETA
        GAMMA
        CONFIG
```

6.5 The VIDICON

The VIDICON TV camera and picture digitizer is a mainstay of the set of visual input devices. The usual facility for taking pictures from the VIDICON is the program CAMERA on ITS (see PART 2 of this USER's GUIDE, W.P. 166). The user who wishes to write an assembly language program which obtains visual data from the vidicon may call the system macro .VIDIN, as follows:

```
 .VIDIN BUFPTR,X,Y  where BUFPTR points to a 10000
        byte picture buffer, and X,Y are the vidicon
        coordinates of the lower-left corner of the
        image to be scanned. The vidicon coordinate system
        ranges from (0,0) in the lower left corner, to
        (511,472) in the upper right corner.
```

7.0 System Maintainance Notes

This chapter is not intended for use by the general user of the mini-robot system. It deals with items of interest to the system programmer or other persons maintaining the system.

7.1 Installing New Users

The procedure for installing a new user on the mini-robot system is as follows:

1) Enter his UIC on the system disk, by logging in as [1,1] and typing

```
:PIP [UIC]/EN
```

Do likewise the new user's disk, mounted on DK1:

```
:PIP DK1:[UIC]/EN
```

2) Enter the new user's name, UIC, ITS login name (if he has none, enter MINI), and his mini-robot mail address (usually the same as his ITS login name -- but NOT MINI), in the file UNAMES.TXT[3,3]. The format for an entry is:

```
UIC
Mini-robot mail address
ITS login name
Full name
```

3) Enter him in the messages/mail directory. Due to the nature of the system interface for messages and mail, this cannot be accessed as a regular file. It is the file MHNDLR.DTY[3,3]. *This file must never be edited or acted upon by PIP, SEND, or RECEIV.* The disk location of this file must remain constant. To read in the directory file, start up a RUG (no file arguments, just type an extra <CR>) and then type:

```
177412/ oldval   14140^
177410/ oldval   50000^
177406/ oldval   -4000^
177404/ oldval   5
```

This sequence reads the directory from disk.
Now, the format of the directory in core is:

```
54776/ current message number
55000/ UIC of first user
55002/ last message number looked at by this user
55004/ undelivered mail item number
55006/      "
55010/ 0

55100/ UIC of second user
55102/ last message number looked at by this user
55104/ undelivered mail item number
  .
  .
  .
55n00/ 0  ; end of table
```

To enter the new user, find the first block of 100 words which begins with a 0 word (current end of table), put his UIC in the first word and make the last message number (second word) equal to or less than the contents of 54776 (this will determine whether he sees any old messages when he first logs in, and if so, how many).

Note that a UIC [n,m] is stored as two bytes: a byte containing n and a byte containing m.

When the table is updated, write it out to disk by typing

```
177412/ oldval   14140^
177410/ oldval   50000^
177406/ oldval   -4000^
177404/ oldval   3
```

(It might be useful to write a program to do this disk I/O and table searching/update).

Note that the files MAIL.TXT and MSGS.TXT on [3,3] contain the current messages and undelivered mail items, but these files may not be edited, as they also contain binary information which EDIT can't handle. MAIL and MSGS should be the only programs to operate on those files.

7.2 Reading diagnostics

Note that there is a system utility for reading in diagnostics from paper tape. Load the paper tape reader, and type

```
:ABSLDR
```

When the tape is read, the processor will halt.

7.3 Formatting a disk pack

The procedure for formatting a disk pack is as follows:

1) Read in the diagnostic called DISK DATA from paper tape

2) LOAD ADDRESS 200

3) Place the number 2003 in the switch register

4) Place the disk to be formatted in the upper drive only

5) Start the processor

APPENDIX A
11/40 START-UP (BOOTSTRAP) PROCEDURE

The procedure for bootstrapping the 11/40 is as follows:

1. Verify that no one else is using the 11/40. If there is someone sitting at the 11/45 console (GT40) ask him, as 11/45 users sometimes use the 11/40 in conjunction with their 11/45 programs. Also, check that the photowriter and photoscanner are not already in use (both should be powered off, unless you just turned the photoscan on yourself). If the photowriter or photoscanner drum is rotating and the lights are blinking, DON'T proceed to step 2.

2. The 11/40 should have power (manifested by red L.E.D.'s showing on the front panel). If it does not, you will have to power up the 11/45 (the two machines are on one power switch) by rotating the three-position rotary switch on the 11/45 front panel from the OFF to the POWER position. You need do nothing else to the 11/45.

3. On the 11/40 front panel, press the ENABLE/HALT switch down (to HALT), then lift it up (to ENABLE).

4. Place the number 173000 in the 11/40 switch register. (Chances are it is already there: all switches down except 15, 14, 13, 12, 10 and 9.)

5. Press LOAD ADDRESS. (It will pop back up.)

6. Press START (it too will pop up), and you're ready to go.

APPENDIX B
11/45 START-UP (BOOTSTRAP) PROCEDURE

The procedure for bootstrapping the 11/45 system is as follows:

1. If necessary, turn the power on. The power switch is a three position rotary switch on the front panel of the 11/45 CPU, with positions labelled OFF, POWER, and LOCK. Turning this switch to the POWER position turns on the computer.

2. Place the RUN/LOAD switch of the uppermost disk drive into the RUN position. Wait for the top three lights to come on. The disk cartridge labelled SYSTEM should be mounted in that drive.

3. Place the HALT/ENABLE switch on the 11/45 main panel in the HALT (down) position. Place the number 177670 into the switch register (switches 15,14,13,12,11,10,9,8,7,5,4, and 3 all up). Press LOAD ADDRESS (it will pop back up), raise the HALT/ENABLE switch (to ENABLE), and press START.

The GT40 screen should now contain the message WELCOME TO THE MINI-ROBOT GROUP on the top of the screen, and a "$". Type a rubout at the GT40 keyboard, and then try typing a character. It should echo on the screen, along with a rectangular cursor. You are now ready to go.

IF THE SYSTEM DOES NOT RESPOND TO TYPED CHARACTERS:

4. Place the RUN/LOAD switch of the uppermost disk drive to the LOAD position, wait about 5 seconds, place it in the RUN position, wait for the three uppermost lights to come on, and go back to step 3.

IF THE SYSTEM STILL DOES NOT RESPOND AFTER EXECUTING STEP 5: Find someone who knows what to do.

IF THE GT40 DOES NOT DISPLAY CHARACTERS, AND THERE IS NO RECTANGULAR CURSOR: Try bootstrapping the GT40. See Appendix C.

APPENDIX C
GT40 START-UP (BOOTSTRAP) PROCEDURE

If in the course of bootstrapping the 11/45 system (Appendix B), the GT40 should be found to not display characters, to not display a rectangular cursor, or to not have its RUN light on, then it will be necessary to boostrap the GT40.

1. On the GT40 (11/05) console, place the HALT/ENABLE switch in the HALT (down) position, and press the START key (this does a RESET).

2. Place the number 166000 in the GT40 switch register. (Bits 15, 14, 13, 11, and 10 all up).

3. Press LOAD ADDRESS (it will pop back up).

4. Raise the HALT/ENABLE switch to the ENABLE (up) position.

5. Press START (it will pop back up).

6. Turn the brightness knob on the GT40 display all the way up. Characters will now display dimly, and there will be no cursor.

7. If you have not already done so, finish bootstrapping the 11/45 system (see Appendix B).

8. When the "$" prompt character appears, type ":DISP"<CR>. Turn the brightness down to about 3/4 of maximum. The word "LOADING" should appear on the screen, and after several seconds, a rectangular cursor should appear.

APPENDIX D
PDP11/45 CORE MAP

-- 11/45 CORE MAP --

INTERRUPT VECTORS

4	- MEMORY PROTECT VIOLATION, ODD ADDRESSING FAULT, STACK OVERFLOW
10	- ILLEGAL INSTRUCTION CODE
14	- TRACE TRAPS (OCCUR WHEN BIT 4 OF PSW IS ON)
20	- IOT TRAPS
24	- POWER FAIL TRAPS
30	- EMULATOR TRAPS (EMT'S)
34	- TRAP INSTRUCTIONS TRAP HERE
60	- TELETYPE INPUT DONE INTERRUPTS
64	- TELETYPE OUTPUT DONE INTERRUPTS
70	- PAPER TAPE READER INTERRUPTS
100	- KW11L LINE CLOCK INTERRUPTS
104	- KW11P PROGRAMMABLE CLOCK INTERRUPTS
124	- DR11B INTERRUPTS
130	- DC11 INTERRUPTS
200	- VIDICON DATA READY INTERRUPTS

```
210     - PHOTOWRITER INTERRUPTS

220     - RK11 DISK INTERRUPTS

300     - DL11 INPUT INTERRUPTS (TERMINET INPUT READY)

304     - DL11 OUTPUT INTERRUPTS (TERMINET OUTPUT DONE)

310     - DR11C #1 (INTERFACE TO 1140)

320     - DR11C #2 (ANALOGIC CONVERTER)

330     - DR11C #3 (BRAKES FOR MECHANICAL ARM)

340     - DR11C #4 (TO PATCH PANEL LABELLED "DR11C INTERFACE")

350     - XY TABLE X MOTION COMPLETE INTERRUPT

354     - XY TABLE Y MOTION COMPLETE INTERRUPT


00500 - 15000  => DOS MONITOR (APPROX.)

15002 - 24000  => RUG SYMBOL TABLE, IF APPLICABLE.
                    ALSO, RESIDENT PORTION OF LISTER, IF APPLICABLE.

24000 - 50000  => RUG, WHEN APPLICABLE.

50000 - 55000  => GTLOAD, WHEN APPLICABLE (WHEN CALLED BY .GTLD)

55000 - 157360  => USER PROGRAM AREA

157360 -157776  => DOS MONITOR USE

160000 - 174000 => SYSTEM USE ONLY (BATCH, IBATCH, BOS, LIST)

174010 - 174016 => X-Y TABLE
```

174020 - 174026	=> VIDICON
174060 - 174066	=> DR11B (UNUSED)
174070 - 174076	=> KW11P PROGRAMMABLE REAL TIME CLOCK (USER CLOCK)
174100 - 174106	=> DC11 (CONNECTED TO ITS VIA T40)
174200 - 174376	=> 64 D/A CHANNELS (ONLY 32 IMPLEMENTED SO FAR)
175750 - 175756	=> DR11C #1 (INTERFACE TO 1140)
175760 - 175766	=> DR11C #2 (CONNECTED TO ANALOGIC D-A AND A-D CONVERTER)
175770 - 175776	=> DR11C #3 (BRAKES FOR MECHANICAL ARM)
176000 - 176006	=> DR11C #4 (PATCH PANEL LABELLED "DR11C INTERFACE")
176500 - 176506	=> DL11 (INTERFACE TO TERMINET)
177400 - 177416	=> RK11 DISK CONTROLLER
177500 - 177506	=> PHOTOWRITER REGISTERS (DA, BA, CT, ST)
177546	=> KW11L LINE CLOCK (SYSTEM CLOCK)
177550 - 177556	=> PC11 PAPER TAPE READER
177560 - 177566	=> SYSTEM TTY (GT40)
177570	=> SWITCH REGISTER (INPUT), DISPLAY REGISTER (OUTPUT)
177600 - 177676	=> ROM BOOTSTRAP LOADER
177776	=> PROGRAM STATUS WORD

APPENDIX E
GT40 CONVENTIONS

This Appendix describes the conventions which the GT40 programmer writing his own scope programs should follow in order for the system software (specifically .GTLD and GTLOAD) to be able to load the GT40 when the user's program is running. These conventions apply only to the programmer writing his own GT40 programs from scratch; the P user need not concern himself with this section, since P observes these conventions.

The user's GT40 program must have an interrupt handler for 11/45 interrupts. The interrupt handler must check for the character ^O (ASCII 17), and take certain actions when this occurs.

For example:

```
300/ DLINT

DLINT:  MOV 175612,R0           ; GET THE CHARACTER IN R0
        BIC #177600,R0          ; CLEAR OFF PARITY
        CMP R0,#17              ; CONTROL-O?
        BEQ ROM                 ; IF SO, START THE BOOTSTRAP ROM

ROM:    RESET
        CLR 15776               ; THE USER MUST CLEAR THIS MAGIC LOCATION
        JMP 166000              ; OR THE ROM WON'T WORK CORRECTLY.
```