306-142
Issue 1

5620 Dot-Mapped Display
Release 2.0
Application Development Guide

TRADEMARKS

The following trademarks are used in this manual:

- *WE* — Registered trademark of AT&T Technologies, Inc.
- *UNIX* — Trademark of AT&T Bell Laboratories
- *TELETYPE* — Registered trademark of AT&T Teletype Corporation
- *DEC* — Registered Trademark of Digital Equipment Corporation
- *VAX* — Registered Trademark of Digital Equipment Corporation
- *PDP* — Registered Trademark of Digital Equipment Corporation

ADDITIONAL COPIES

To order additional copies of this document, call the AT&T Customer Information Center at: **1-800-432-6600** (toll free) and ask for "Select Code 306-142".

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

THIS GUIDE AND YOU

You must be familiar with the C programming language and the UNIX* Operating System to understand this guide.

For your convenience, we have arranged each chapter with an introduction to guide you to the level of detail you need. The index and the table of contents in each chapter are quick "pointers" to the specific topic of interest.

After you are familiar with TELETYPE† 5620 Dot-Mapped Display terminal (DMD) programming, you can use the *5620 Dot-Mapped Display Reference Manual* (306-144) to find data in the familiar "manual page" format.

---

* Trademark of AT&T Bell Laboratories

† Registered Trademark of AT&T Teletype Corporation

This guide is arranged as follows:

CHAPTER	DESCRIPTION
Introduction	Contains introductory material.
Programming the DMD	Describes DMD programming details.
Dmdebug	Describes the DMD program debugger, **dmdebug**.
Programming Layers	Describes the "**-f**" and "**hagent**" features of **layers**.
Icon	Describes the DMD icon and texture editor, **icon**.
The DMD Compiler	Describes the DMD C language compiler, **dmdcc**.
Utilities	Describes the Software Generation System (SGS) tools provided with the DMD Application Development Package.
Appendices	Appendix A lists the vector tables for DMD firmware. Appendix B contains sample source code for several DMD demo programs. Appendix C lists the reserved words for DMD software.

COMMENTS

If you have any comments, suggestions, or ideas, please take a moment to fill out the comment card in the back. Your help is welcome.

FURTHER READING

You can order the AT&T documentation catalog, which has a listing of all our documents, by calling our toll-free order number:

1-800-432-6600.

Chapter 2

PROGRAMMING THE DMD

PAGE

Chapter 2

PROGRAMMING THE DMD

This chapter is a tutorial. Read all of it to gain an understanding of DMD programming. If you need "brushing up," the table of contents points to specific topics.

DMD ENVIRONMENTS

There are two compatible programming environments for the DMD. One is the **stand-alone** environment, where the program running in the DMD is the only one running in the terminal. The other is the time-shared environment, **layers**, in which several programs multiplexed in time and screen-space coexist.

The **layers** environment is a parallel, transparent, error-corrected connection between the host and DMD processes. The **stand-alone** environment has no inherent protocol, and programs must generate a protocol to prevent communication errors.

Only the **layers** environment inhibits cursor and display operations. Graphics in the **stand-alone** environment must inhibit the cursor to avoid interfering with the mouse.

The following functions and globals are available only in the **layers** environment:

canon	**getrect**	**itox**	**outline**
P	**physical**	**alarm**	**realtime**

The debugger **dmdebug** and the icon editor **icon** only run in **layers**.

COMPILING

To compile a standard C program, type: **dmdcc** *source.c* where *source.c* is the name of the C language source file. The **dmdcc** default compiles a program to run under **layers**. The flag **-J** invokes the **stand-alone** environment. Refer to Chapter 6, "THE DMD COMPILER," for more information on options to **dmdcc**.

DMD programs can run in both the **layers** and **stand-alone** environments by using the following C language construct:

```
#ifdef MPX
....   /* The code that runs in layers */
#else
....   /* The code that runs in stand-alone */
#endif
```

MPX is defined if compiled for the **layers** environment and undefined if compiled for the **stand-alone** environment. The environment dependent code is placed within the if-else statement. The appropriate section of code is selected at execution.

Note: Contact your DMD software administrator (usually your UNIX System Administrator) for the path to the DMD C compiler (**dmdcc**), linking loader (**m32ld**), and assembler (**m32as**).

PROGRAM EXECUTION

The DMD has a number of Erasable Programmable Read Only Memory (EPROM) resident programs. These include the resident terminal program, a boot loader, library subroutines, and the terminal side of **layers**. The resident terminal program is executed when the DMD is powered on (see Figure 2-1).

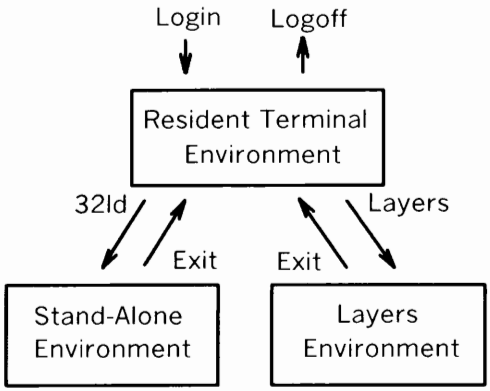


Figure 2-1. DMD Environments

The UNIX System host program, **32ld**, downloads programs into the DMD. The **32ld** loader has two types of downloads: The first is a stand-alone download executed from the resident terminal environment. The second is a download of programs into a layer. When the terminal program receives an "**ESC** [**v** " character sequence from **32ld**, it executes the boot loader. The boot loader clears the screen and waits for the UNIX System host (via

32ld) to send the program to the DMD. The program is copied into DMD Random Access Memory (RAM) starting in low memory. The boot loader sets the display to point to low memory, so the incoming program can be viewed during loading. Downloads into a layer work in a similar manner, except the 'coffeecup' cursor is displayed and the layer is filled from bottom to top with inverse video as the program loads.

> ***Note:*** Booting **layers** is a special case of the **stand-alone** download. There may or may not be a download from the host, depending upon the version of the firmware.

Normally, a C language program requests Input/Output (I/O) from the system running it. The DMD, however, does not maintain files in its memory, though processes do run in the DMD. The DMD I/O interpreter, **jx**, resolves the problem by forcing standard I/O requests to be handled by the UNIX System host. The **jx** I/O interpreter runs in the UNIX System host waiting for I/O requests from the DMD. Refer to the section on "**jx**" under "Resources and I/O" in this chapter.

The UNIX System program **jx** downloads programs to run in an environment which includes a Standard I/O interpreter (obtained by **#include <dmdio.h>**). If the program does not use Standard I/O, or some other protocol is desired, the program should be loaded directly with **32ld**, not **jx**. Refer also to **jx(1)** in the *5620 Dot-Mapped Display Reference Manual* for more information.

To restart the DMD after a software failure, press the SHIFT and SETUP keys simultaneously, then press DISCON/BRK (break). This will reset the DMD. See your *5620 Dot-Mapped Display Terminal Owner's Manual* for more details.

When a layer is created, a terminal process is run in the layer, copying UNIX System characters to the screen and sending keyboard characters back to the standard input of the UNIX System process (for example, the UNIX System shell or text editor). If a UNIX System process (usually **32ld**) sends the ioctl JBOOT, the boot loader is invoked for that layer, and

a program compiled for the **layers** environment is loaded and run in the layer. If that process calls **exit( )**, or if the ioctl JTERM is sent, the layer will automatically restart the window program for that layer.

See **layers(1)** in the *5620 Dot-Mapped Display Reference Manual* for more information; the rest of this guide assumes some understanding of its workings.

SPLITTING THE DMD PROGRAM

The DMD is an intelligent terminal, not a personal computer. A significant program using the DMD must have some support from the host UNIX System, for example, to access system resources (such as files). In some cases, a DMD program requires two separate programs and a protocol for their communication (for example, the DMD text editor, **jim**). This division has advantages: it allows a clean division of work between two processors, which can lead to a clean, simple design; and it results in a smaller program running in the terminal, which downloads faster and consumes less memory. But the division must be made, and it can be very difficult to choose a good place to divide. The low bandwidth connection imposes the further constraint that communication across the division should be tightly encoded. The **jx** I/O system can be an easy solution by simulating standard I/O functions in the terminal program. But that is not always a sensible approach.

For example, a text editor is not practical with **jx** because a context search requires looking at the entire file, which is read by the program using the low-speed RS-232 line. A better approach for an editor is to write a UNIX System program to do file I/O and associated functions. A UNIX System program does file I/O more quickly. Let the DMD program manage the display and user interactions.

DATA TYPES

This section defines the data types in **$DMD/include/dmd.h**.

> ***Note:*** Refer to **structures(3)** and **menuhit(3)** in the *5620 Dot-Mapped Display Reference Manual* for detailed information.

Word

A **Word** is a 32-bit integer:

```
typedef int Word;
```

It is the quantum of memory used by the graphics software.

Point

A **Point** is a structure containing two short integers (16 bit), specifying a point on the screen or in a bitmap:

```
typedef struct{
        short x;
        short y;
}Point;
```

The coordinate system is oriented with X positive to the right, ranging from 0 to XMAX-1 across the screen, and Y positive down, ranging from 0 to YMAX-1 from top to bottom. (Where XMAX = 800, YMAX = 1024.)

Rectangle

A Rectangle is specified by two **Point**s:

```
typedef struct{
        Point origin;    /* Upper left */
        Point corner;    /* Lower right */
}Rectangle;
```

The **origin** is the location of the upper left corner (minimum X and Y) of the **Rectangle**, and **corner** is the lower right (maximum X and Y). By convention, the right (maximum X) and bottom (maximum Y) edges are excluded from the rectangle, so abutting rectangles have no points in common. This is why there are -1's in the ranges of X and Y across the screen. The **Rectangle** describing the screen is {0, 0, XMAX, YMAX}. These coordinates are screen coordinates.

Bitmap

A **Bitmap** is a storage area for a rectangular image, defined by the enclosing rectangle, and the storage itself:

```
typedef struct{
Word *base;
  /* pointer to start of data */
unsigned width;
  /* width in words of total data area */
Rectangle rect;
  /* rectangle in data area, screen coords */
char *_null;
  /* unused, always zero */
}Bitmap;
```

A **Bitmap** may only have pixel values zero and one and is, therefore, incapable of grey-scale or color drawings. By turning on pixels in a regular pattern, you may draw textures that may be used for many of the same functions as color on color displays.

Bitmap storage is arranged just like a two-dimensional array, with X varying faster.

base
: points to the word of storage containing the upper left corner of the **Bitmap**. The following words contain the rest of the uppermost scan line of the image. In memory, the last word of one scan line is followed immediately by the first word of the next scan line.

width
: is the number of words of storage consumed by a scan line.

rect
: is the coordinate system inside the **Bitmap**.

rect.origin
: is the coordinate of the upper left point in the **Bitmap** image. The coordinates may be any positive or negative short integers. Graphics operations on a **Bitmap** are clipped to **rect**. **Rect** is often the screen coordinates of the rectangle saved in the **Bitmap**.

display
: is a global **Bitmap** describing the screen area available to the program.

Textures

A **Texture** is a 32 by 32 rectangle of bits which defines a dot pattern. For example, a 'tweed' **Texture** is declared as:

```
Texture tweed={
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222,
  0x44444444, 0x77777777, 0xEEEEEEEE, 0x22222222
};
```

A **Texture** looks much like a 32 by 32 **Bitmap**: the first word is the first horizontal scan of the texture, the second is the next, and so on. The routines which use textures fix the patterns to the absolute screen coordinates so that, for example, if two overlapping screen rectangles are textured with the same **Texture**, the dots in each rectangle will mesh properly to form a constant pattern.

The simplest way to visualize how textures are implemented is to recall the off-screen **Bitmap** described earlier, and imagine the array replicated to cover the entire area of the **Bitmap**. Then, rectangles may be textured by copying the corresponding bits, again according to a **Code** argument, from the secondary **Bitmap** to the target **Bitmap** or screen.

By this definition, you may texture images identically by dividing them into any set of rectangles. Since the texture is bound to the coordinates of the secondary **Bitmap**, the texture will be replicated smoothly across the image. For example, the display area may be textured by:

```
texture(&display, Drect, &tweed, F_STORE);
```

but the following (much slower) code will produce the same result:

```
register i, j;
for(i=Drect.origin.x; i<Drect.corner.x; i++)
  for(j=Drect.origin.y; j<Drect.corner.y; j++)
    texture(&display, Rect(i, j, i+1, j+1),
      &tweed, F_STORE);
```

Refer to "Global Structures" in this chapter.

A few **Texture**s are used continuously by DMD programs. These are stored in DMD ROM for fast access. They are:

T_background	**T_darkgrey**
T_grey	**T_lightgrey**
T_white	**T_black**
T_checks	

Refer to Chapter 5, "ICON" for more information on textures provided with DMD software, in the section "SUPPLIED ICONS".

Some **Texture16**s are also stored in DMD ROM. Refer to **Texture16** (next) for a list.

Texture16

A variation of **Texture** is **Texture16**, a 16 by 16 bitmap. It is used to define mouse cursors and graphics icons. The AT&T logo is the sample Texture16 below.

```
Texture16 globe = {
    0x07E0, 0x0000, 0x207C, 0x7FFE,
    0x0000, 0x803F, 0xFFFF, 0x0000,
    0xC07F, 0xFFFF, 0x0000, 0x7FFE,
    0x7FFE, 0x0000, 0x1FF8, 0x07E0,
};
```

Most **Texture16**s are stored in **$DMD/icon/texture16**. A few **Texture16**s are stored in DMD ROM. They are:

C_crosshair	**C_clock**
C_move	**C_skull**
C_sweep	**C_target**
C_confirm	**C_cup**
C_deadmouse	

For more detailed information on using **Texture16**s, see the **icon** discussion in Chapter 5, "ICON," and **icon(1)** in the *5620 Dot-Mapped Display Reference Manual.*

Menu

The routine **menuhit** presents the user with a menu and gets the menu selection. It is declared:

```
int menuhit (m, n)
Menu *m;
int n;
typedef struct Menu {
    char     **item;
             /* string array, ending with 0 */
    short    prevhit;
             /* retained from previous call */
    short    prevtop;
             /* also retained from previous call */
    char   *(*generator)( );
             /* used if item == 0 */
} Menu;
```

Refer to **menuhit(3)** in the *5620 Dot-Mapped Display Reference Manual*.

The routine **menuhit** presents the user with a menu specified by the pointer '**m**' and returns an integer indicating the selection made, or -1 for no selection. '**n**' specifies which mouse button to use for the interaction (button 1, 2, or 3). **Menuhit** assumes that the mouse button is already depressed when it is called. You make a selection by releasing the mouse button when the cursor points to the desired selection; releasing the mouse button outside the menu indicates no selection.

The maximum number of menu items displayed at any one time is 16. When the number of items is 16 or less, all of the items are displayed and are centered one entry per line in the menu. This is the normal menu mode.

When there are more than 16 menu items to be displayed, the menu becomes a *scrolling* menu. The left portion of the menu contains a scroll bar used for scrolling quickly through the menu selections.

Mouse

The global **mouse** is defined:

```
struct Mouse {
    Point xy;
    Point jxy;
    int   buttons;
} mouse;
```

The global location **mouse** contains the current mouse coordinates and button states. The point **xy** is in screen coordinates. The point **jxy** is in layer coordinates. The **buttons** are most easily interpreted using the button macros. See **buttons(3)** in the *5620 Dot-Mapped Display Reference Manual* for detailed information.

GRAPHICS

A DMD program must begin

```
#include <dmd.h>
```

to define DMD data types. Programs using **jx** must begin

```
#include <dmd.h>
#include <dmdio.h>
```

to define DMD data types and the **jx** I/O protocol.

DMD Coordinates

Because a program may be running **stand-alone** using the whole screen, or running under **layers** confined to a rectangular portion of the screen, there are two coordinate systems: *screen coordinates* and *layer coordinates*. Screen coordinates refer to the actual pixels of the screen: (0, 0) is the upper left corner, and (XMAX-1, YMAX-1) is the lower right corner of the screen.

Layer coordinates refer to the rectangular portion of the screen actually used by the program. They are scaled so that (0,0) is the upper left corner, and (XMAX-1,YMAX-1) is the lower right corner of the screen area available to the program. (This portion of the screen is called the program layer.) Layer coordinates are scaled; therefore, the adjacent locations in layer coordinates do not necessarily refer to separate screen pixels.

Global Structures

Several global structures describe the screen portion available to the program:

display
: is a **Bitmap** that defines the display available to the layer.

display.rect
: defines the rectangle (including the border) surrounding the layer.

Drect
: The global rectangle **Drect** defines the screen area inside the border in screen coordinates.

Jrect
: The rectangle **Jrect** is defined as {0,0,XMAX,YMAX} and describes the screen area inside a layer in layer coordinates (including the border).

physical
: is a **Bitmap** describing the entire screen in screen coordinates.

Most of the graphics routines take their arguments in screen coordinates and require target **Bitmap**s to be explicitly passed (by reference, never by value). Those routines whose names begin with a '**j**' take layer coordinates and operate on **display Bitmap**s. Many programs always want to have graphical operations scaled to fit the layer, so the layer appears to the program just like a full screen. Such programs use the '**j**' routines. These

routines also remember a current point (much like dot in ed) which make it simple to draw a curve, for example, by a set of line segments. A user application program can also access this current point position by referring to the global variable **Point PtCurrent**.

Programs dealing largely with text or offscreen **Bitmap**s must be careful how objects appear in the layer and must work (at least sometimes) in screen coordinates, avoiding the '**j**' routines. In general, both layer and screen coordinates will be used, but the careful programmer will be able to write a program that can run unchanged either in **stand-alone** or in **layers**. Except for the coordinate differences and internal details of implementation, the **stand-alone** and **layers** environments are very similar, and all the routines have the same specifications in either environment (unless noted otherwise).

Note: Every DMD program must include <**dmd.h**> to define the basic data types.

Storage Codes

Most graphics routines take a **Code** argument to specify the logical function to use for drawing. The values and meanings of a **Code** are:

F_OR	target	|=	source
F_CLR	target	&=	~source
F_XOR	target	^=	source
F_STORE	target	=	source

In other words, if a rectangle is copied to another place with **Code** F_OR, the result will be the bitwise OR of the source rectangle and the contents of the target area. For line-drawing and points, F_STORE and F_OR are usually equivalent. F_STORE is generally more useful for copying rectangles and drawing textures.

Code F_XOR is the bitwise exclusive OR of the source rectangle with the contents of the target area.

The following program example draws a grid with spacing defined by XSPACING and YSPACING.

```
#include <dmd.h>
#define  XSPACING  25
#define  YSPACING  25

main()
{
  Point p;

  /* Vertical lines */

  for(p.x=p.y=0; p.x<XMAX; p.x+=XSPACING)
      jsegment(p, Pt(p.x, YMAX), F_OR);

  /* Horizontal lines */

  for(p.x=p.y=0; p.y<YMAX; p.y+=YSPACING)
     jsegment(p, Pt(XMAX, p.y), F_OR);
}
```

Since the program uses **jsegment** to draw the lines, the total number of lines will be the same regardless of the layer's shape. The function **Pt(x,y)** passes the point (x,y) to **jsegment**, in lieu of structure-valued constants in C language. **Pt** and its relatives **Rect** (Rectangle from four coordinates) and **Rpt** (Rectangle from two Points) only work in parameter lists.

The grid program draws the lines in F_OR mode, which simply turns on all the bits in the line. The grid may be erased again by executing the same **Code** in F_CLR mode, which turns all the bits off. It may also be erased by clearing the screen, using either

```
jrectf(Jrect, F_CLR);
```
or
```
rectf(&display, Drect, F_CLR);
```

For many graphics operations, F_OR and F_CLR are inverses, since one is the opposite of the other. It is not true, however, that following an F_OR operation with an F_CLR operation will undo the first, since the F_CLR will clear all bits, even those that were set before the F_OR.

F_XOR (with the above commands) inverts each target bit, turning ones to zeros and zeros to ones (all the modes have slightly different meanings in **bitblt** and **Texture**; we shall return to these later). F_XOR has several useful properties:

- F_XOR is its own true inverse: two adjacent identical F_XOR operations cancel exactly, restoring the screen to its previous form.
- F_XOR is commutative: F_XOR operations may be executed in any order to produce the same final result. Combined with its inverse property, this means that an F_XOR operation may be canceled at any later time by another F_XOR operation.
- Because F_XOR is its own inverse, the same code can be used to draw or undraw a picture. This is a common action: the mouse cursor, for example, is updated by calling the same routine, using F_XOR mode internally, to undraw the old position and draw the new one (the order is irrelevant!).
- Application programs should avoid writing to screen bitmaps other than the **display** bitmap in order to avoid interference with the mouse cursor ("mouse tracks") when not using F_XOR mode.
- Because the mouse cursor is tracked in F_XOR mode, any operations done in F_XOR mode will never interfere with the cursor. But, the graphics routines must inhibit the cursor to avoid leaving "mouse tracks" when drawing with other function codes. (This applies to the **stand-alone** environment only.)

Of course, these properties can break down if F_XOR operations are mixed with other modes. However, it is not only possible, but common and even natural, to do all graphics in F_XOR mode. F_XOR can also be used to simulate the video bit: inverting the screen or some portion of it with a call such as:

```
jrectf(Jrect, F_XOR);
```

changes the sense of all subsequent and previous F_XOR operations.

bitblt

The routines **point**, **segment**, and **rectf** define a set of points that perform an action defined by the **Code** argument (such as F_XOR). There is another, more fundamental, way to define these routines which both explains the names of the **Code**s and introduces the **Bitmap** operator, **bitblt**. Graphical operators are defined as a secondary off-screen **Bitmap**. They are set to zeros everywhere except the points corresponding to the points on the screen where the operation is performed. This includes the points of the approximating line of **segment**, the single point of **point**, or the points in the rectangle of **rectf**. The routines then perform a bit-for-bit logical function from the secondary **Bitmap** into the screen (or whatever target **Bitmap** has been specified):

F_OR:	screen location	|=	secondary location
F_CLR:	screen location	&=	~secondary location
F_XOR:	screen location	^=	secondary location
F_STORE:	screen location	=	secondary location

The operator to do this is **bitblt**, although (for efficiency rather than practicality) some of the basic graphics functions (such as the line and point-drawing routines) are written without it.

Bitblt is actually more general, doing the bit-for-bit function from an arbitrary **Rectangle** in an arbitrary **Bitmap** to an arbitrary **Rectangle/Bitmap** destination. It is declared:

```
bitblt(sourcemap, sourcerect,
       destmap, destpt, code)
  Bitmap *sourcemap, *destmap;
  Rectangle sourcerect;
  Point destpt;
  Code code;
```

sourcerect and **destpt** are in the coordinate systems of their particular **Bitmaps**. The point **destpt** specifies the **Point** corresponding to the origin of **sourcerect** in the destination **Bitmap**; the corner is derived from the destination **Rectangle** being congruent to the source. The source and destination **Bitmaps** may be the same, and the source and destination rectangles may even overlap. **Bitblt** always does the assignments in the correct order.

Here, for example, is how to scroll the screen up 16 pixels:

```
/* copy everything up 16 pixels */

bitblt(&display, Rect(0, 16, XMAX, YMAX),
  &display, Pt(0, 0), F_STORE);

/* now clear the bottom line */

rectf(&display, Rect(0, YMAX-16, XMAX, YMAX), F_CLR);
```

Note 1: This code assumes it is running **stand-alone**, and can scribble freely on the screen.

Note 2: There is no **jbitblt** -- **bitblt** operates directly on **Bitmaps**, which are specified in screen coordinates.

A common use of **bitblt** is to copy a prepared picture (such as a character) from off-screen onto the display. The demonstration program "ball" (in "APPENDIX B") simulates a bouncing ball by building an off-screen **Bitmap** and using **bitblt** to make it bounce.

> ***Note:*** All DMD library routines are explained in the *5620 Dot-Mapped Display Reference Manual.*

RESOURCES AND I/O

Input/Output from a DMD program is totally different from I/O in a UNIX System program. The differences are inherent, essential, and invaluable, and all stem from the DMD being a terminal rather than a computer running a UNIX System process.

A program running in a DMD does not have standard input or standard output. For example, the DMD is connected to a host UNIX System by a single RS-232 connection; a bidirectional, low-speed link.

As a terminal, the job is twofold: first, send characters typed on the keyboard down the RS-232 link to the standard input of a UNIX System process; and second, receive characters sent from the standard output of a UNIX System process and draw them on the screen. The ideas of standard input and output are meaningless in this isolated environment. Characters requiring attention may appear at any time from either the host or the keyboard, so the DMD must be prepared to deal with two I/O devices at any time. A UNIX System "read" cannot read from two file descriptors at once. So, if it reads from the keyboard, it may miss characters sent from the host. Generally, DMD I/O requests never block (wait for a resource to become available). Instead, they return an error status if the request cannot be serviced. A DMD program may, however, explicitly ask to wait for a resource to become available.

A DMD program has access to these I/O resources:

RCV
Characters received from the UNIX System

SEND
Characters sent to UNIX System

KBD
Characters typed on the keyboard

MOUSE
The graphics input device

CPU
The DMD central processing unit (a pseudo resource)

ALARM
A pseudo resource

PSEND
Send a character to the printer.

A program wishing to use one or more of these devices indicates its intention by a **request( )** call. Multiple resources may be allocated by OR'ing the resource names together; for example

```
request(KBD|MOUSE);
```

allocates the keyboard and mouse to the program. A **request( )** call overrides all previous calls -- all desired resources must be requested in one call.

Characters sent to and from the UNIX System are kept in two queues: the RCV queue for characters coming from the UNIX System to the program, and the SEND queue for characters going to the UNIX System (see Figure 2-2 on the next page).

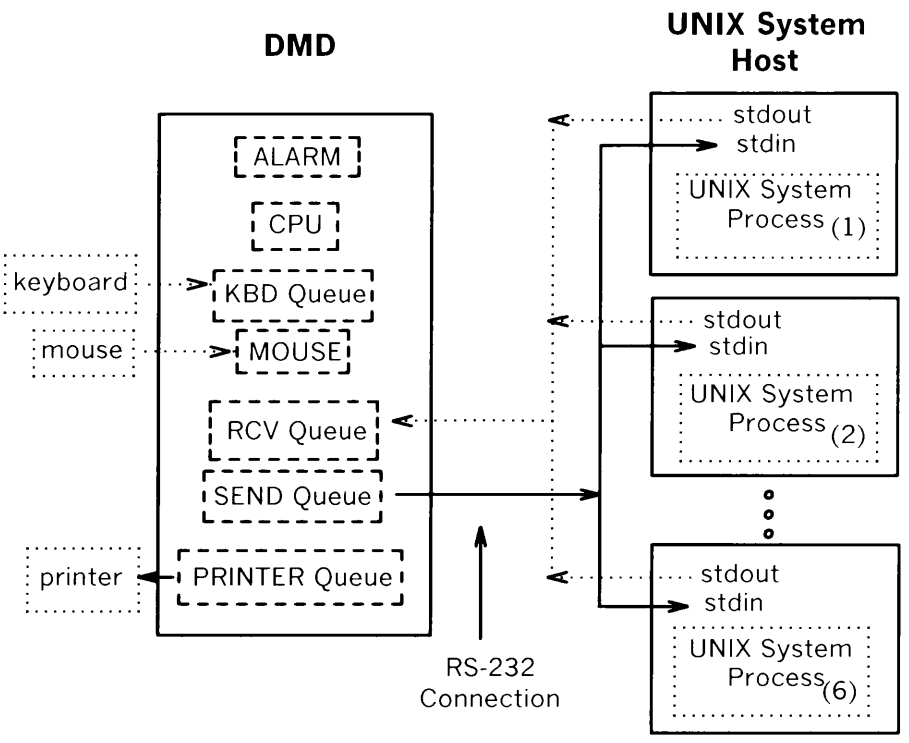


Figure 2-2. DMD Resources

Characters read from the RCV queue are typically written on the standard output or standard error of the UNIX System process associated with the layer. Characters put on the SEND queue are sent to the UNIX System and may be read on the standard input of the associated UNIX System process. Note the terms "standard input," "standard output," and "standard error" apply to the UNIX System process, not the process running in the DMD. The RCV and SEND queues in the DMD are similar to standard input and output, but RCV characters come from the UNIX System standard output, and SEND characters go to UNIX System

standard input. In other words, the DMD and UNIX System processes have I/O cross-coupled, much like two UNIX System processes connected by a pair of pipes.

The routine **rcvchar( )** reads and returns the next character from the process's RCV queue, or -1 if the queue is empty. **Sendchar(c)** similarly puts the character "c" on the SEND queue, and thereby transmits it to the UNIX System. **Sendnchars(n,p)** similarly transmits the "n" characters pointed to by **p**.

If the keyboard has been requested, **kbdchar( )** will return the next character typed to the process on the keyboard, or -1 if no characters are outstanding. Typed characters are held in a queue until KBD is requested, or the program exits. In the **stand-alone** environment, the characters are thrown away if the keyboard is not requested.

The standard **layers** terminal program, for example, does not request the keyboard; typed characters are simply sent to the UNIX System process. But many programs (for example **jim**, the DMD text editor) wish control over the keyboard.

The semantics of mouse allocation are a little more complicated, and relate to the user interface of **layers**. **Layers** has a current layer — at most one layer which currently owns the keyboard and mouse. Clicking button 1 in a layer, or using the "Current" menu option on button 3, hands both the keyboard and mouse to the process in the indicated layer. Pressing a mouse button in the current layer passes the button hit to the layer if it has requested the mouse; otherwise, it is interpreted by the DMD. For example, **button2( )** is true only when: the process is in the current layer, the mouse cursor points to a visible portion of the layer, the process has requested the mouse, and button 2 is depressed. The next section explains the interface to the mouse in more detail.

Two primitives, **own( )** and **wait( )**, inform a process about its resources. **Own( )** returns a bit vector of the requested resource status:

own ( )&RCV
True if the RCV queue has a character, and RCV is requested.

own ( )&SEND
Always true.

own ( )&KBD
True if the KBD queue has a character, and KBD is requested.

own ( )&MOUSE
True if you own the mouse.

Note that ownership of the mouse is independent of the status of the buttons; basically it implies that the system will pass to the current process mouse coordinates and button status as they change. (See the next section for more information about the mouse structure.) A process never owns a resource it has not requested. (SEND is always implicitly requested.)

A process owns the mouse only if all the following are true:

- The process has requested the mouse.
- The process's layer is the current layer.
- The cursor is within the boundaries of the process's layer, and the layer is not covered.

For example, a typical use of **own( )** is to see which of several I/O devices need service:

```
main( ){
   int got;
   request(KBD|RCV);
   for(;;){
         got=own( );
         if(got&KBD)
               kbdservice( );
         if(got&RCV)
               rcvservice( );
             ...
    }
}
```

DMD Scheduling

The DMD **layers** environment does not do pre-emptive scheduling, so processes must explicitly give up the CPU to enable other processes to run. The function **wait( )** resolves this difficulty by suspending a process until a resource becomes available. **Wait( )** takes an argument resource bit-vector and waits until at least one of the resources becomes available, then returns the requested resource. Thus, the **own** example at the end of the (previous) **bitblt** discussion could be written:

```
main( ){
   int got;
   request(KBD|RCV);
   for(;;){
     got=wait(KBD|RCV);
       if(got&KBD)
           kbdservice( );
       if(got&RCV)
           rcvservice( );
       ...
   }
}
```

Some programs, such as game programs, wish to run continuously. To enable other processes to run, a fake resource, CPU, which is always implicitly requested, may be waited for. **Wait(CPU)** enables all other processes which are ready to run, and returns immediately after they have had a chance to run, regardless of the state of other resources. The inner loop of a game program might look like:

```
for(;;){
  wait(CPU);
  updatedisplay();
}
```

The Mouse

The software does automatic cursor tracking and keeps relevant variables about the mouse in a global structure called **MOUSE**. For example, the following is a C program to draw a curve sketched by the mouse whenever button 1 is depressed and exit when button 3 is depressed:

```
#include <dmd.h>
main()
{
  request(MOUSE);
  for(;;){
    wait(MOUSE);
    if(button1()){
  y);
        for(; button1(); nap(2))
          jlineto(mouse.jxy, F_OR);
    } else if(button3())
          exit();
  }
}
```

Mouse.xy is the mouse position in screen coordinates; **mouse.jxy** is in layer coordinates. The '**button**' macros are true when the corresponding buttons are depressed and the mouse is in the program's layer. The macro **button12( )** is true when either button one or two is depressed, and similarly for **button23( )** and **button123( )**. **Nap( )** suspends the process for some number (the argument) of 'ticks' of the video refresh. A tick is approximately 1/60th of a second. The display is interlaced, so sleeping for two ticks guarantees a screen update before proceeding.

Jx

The DMD execution and **stdio** interpreter is **jx**. The **jx** syntax is
`jx` *file* where *file* is the downloaded portion of a DMD program using **jx** for I/O.

Programs using **jx** should begin

`#include <dmdio.h>` and terminate with `exit( )`.

Jx downloads the program in *file* to the DMD on **/dev/tty***xx* and runs it there, simulating most of the standard I/O library functions. **Stdout** and **stderr** are properly redirected, while **stdin** is properly redirected only if it is not from the keyboard. Programs wishing to read from the keyboard should use the **kbdchar( )** library routine.

If data is directed to **stdout** or **stderr**, **stdout** appears in **$HOME/.jxout**, and **stderr** appears in **$HOME/.jxerr**.

At **exit( )**, control is returned to the shell and the resident terminal program is rebooted.

Note: For **stand-alone** operation, programs using **jx** should be compiled with the **-J** option of **dmdcc**.

Remember these points when using **jx**:

- **Stdout** goes to **$HOME/.jxout**.
- **Stderr** goes to **$HOME/.jxerr**.
- Compiled programs must use **exit( )** to terminate execution.
- Compiled programs must start with #**include** <**dmdio.h**>.
- Do ***not*** use **sendchar**, **sendchars**, and **rcvchar**.

The **stdio** functions available from **jx** include:

access	**fclose**	**fflush**	**fgets**
fopen	**fprintf**	**fputs**	**fread**
freopen	**fwrite**	**getc**	**getchar**
pclose	**popen**	**printf**	**putc**
putchar	**puts**	**sprintf**	

CHARACTERS AND FONTS

The DMD has access to multiple character sets or fonts. A font is internally a single **Bitmap**, with some associated information. The **Bitmap** contains the bit pattern for each character, arrayed adjacently into a long horizontal stripe:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

It is not necessary that the characters appear in ASCII order, but for convenience, most do. The characters in the **Bitmap** are all aligned on the same baseline.

The font structure is defined in <**font.h**>:

```
typedef struct
{
  short n;       /* ascii value of last char in font */
  char height;        /* height of bitmap */
  char ascent;        /* top of bitmap to baseline */
  long unused;  /* for a rainy day */
  Bitmap *bits;       /* where characters are stored */
  Fontchar info[n+2]; /* n+2 character descriptors */
}
Font;
```

The extra **Fontchar** is present to indicate the right edge of the (n+1)th character. A single **Fontchar** structure describes each character:

```
typedef struct
{
  short x;
        /* left edge of bits in bitmap */
  char top;
        /* y of first non-zero scan-line */
  char bottom;
        /* y of last non-zero scan-line */
  char left;
        /* x offset of baseline */
  char width;
        /* width of baseline */
}
Fontchar;
```

To draw a character on the screen requires transferring the appropriate rectangle from the font **Bitmap** to the screen at the correct location. For F_OR, F_XOR and F_CLR modes, the minimum enclosing rectangle of the character is all that needs transferring. For F_STORE mode, the entire target rectangle, the size of a complete character, must be copied from the Font **Bitmap** to the destination **Bitmap**.

> ***Note:*** < **font.h** > must be included in any program that uses the **Font** or **Fontchar** structures.

The example on the next page shows the routine to draw a character **c** from **Font fp** in a **Bitmap db** at **p** with **Code f**.

```
#define sext(val) ((val & 0x80)? val | 0xFFFFFF00: val)

/* the macro sext does character sign extension */

#include <dmd.h>
#include <font.h>
drawchar(c, fp, db, p, f)
  char c;
  Font *fp;
  Bitmap *db;
  Point p;
  Code f;
{
  Rectangle r;
  Fontchar *i=fp->info+c;
  p=transform(p);
/* convert from layer to screen coordinates */
  if(f == F_STORE)
    {
      r.origin.y = 0;
      r.corner.y = fp->height;
    }
  else
    {
      r.origin.y = i->top;
      r.corner.y = i->bottom;
    }
  r.origin.x = i->x;
  r.corner.x = (i+1)->x;
  bitblt(fp->bits, r, db,
  Pt(p.x + sext(i -> left),
  p.y + r.origin.y),f);
}
```

Note that the coordinate system places the origin of the tallest character at the specified point, not at the baseline. This behavior is consistent with the coordinate system of Rectangles but requires some programming if characters from several fonts are placed on the same baseline.

The DMD default font set (**defont**), resident in the DMD ROM, is used by resident terminal programs. Several service routines give simple access to

fonts stored on the host file system. The *5620 Dot-Mapped Display Reference Manual* describes all the library routines.

VECTOR TABLES

Overview

There are two vector tables used to link downloaded software with ROM-resident library routines and associated data structures. These tables are declared as arrays of integer pointers. One table (called the "Stand-alone Vector Table") is used in the **stand-alone** environment; the other (called the "Layers Vector Table") is used in the **layers** environment. APPENDIX A contains a list of the vector table entries. The **stand-alone** vector table is located in ROM, at absolute address 0x200. The **layers** vector table is copied into RAM when layers starts; it begins at absolute address 0x71d700.

Using the Vector Table

The vector table is used to call any ROM-resident routine from a downloaded program. In addition, references to firmware data structures (see APPENDIX A) are also made indirectly through the vector table.

C language programs compiled with **dmdcc** will automatically use the vector table. Assembly language programmers will need to create their own indirect calls as described below.

Not all entries in the vector table are intended for use by the casual user. Application programmers should restrict themselves to using only those routines and data structures described in the *5620 Dot-Mapped Display Reference Manual*.

Caution: The functionality of undocumented routines and data structures is not guaranteed. The calling sequence and other details may change in the future.

Implementation Details

Dmdcc Linkage

C language programs that include **dmd.h** will have all their references to names in the vector table translated into indirect references through the vector table. This makes the resulting executable independent of firmware changes in the terminal.

Note: You must be careful not to use the names in the vector tables for their own data structures or routines. These names should be considered reserved words by DMD programs (see APPENDIX C, "Reserved Words").

The translation is actually accomplished by **cpp**(1), the C language preprocessor. The names of the vector table functions are defined as macros that expand into indirect references through the vector table. The C language preprocessor will substitute the macro expansion for later occurrences of the names of routines and data structures in the vector table. Two include files are used: **sa.h** is used for **stand-alone** programs, and **mpx.h** is used for programs targeted for the **layers** environment. The symbol **MPX** is defined by **dmdcc** when the program is being compiled for the **layers** environment. **dmd.h** includes the appropriate jump table linkage file, **sa.h** or **mpx.h**, based on whether **MPX** is defined.

References are cast by the macros to the appropriate type of the function return value or data structure.

Assembly Language and the Vector Tables

Most C language programmers will never realize that the vector table is being used for some of their library routine and data structure references; it happens transparently as part of the compilation process. However, assembly language programmers will have to explicitly code indirect references.

Assembly language programmers should consult APPENDIX A for the list of routines in the vector table. An indirect call can be generated as shown below: (the example is for 'qputc' in the firmware vector table)

```
                .
                .
        call    &1, *(80*4+Firm)
```

or

```
        .set    Firm, 0x200
        .set    qputc, 80*4+Firm
                .
                .
        call    &1, *qputc
```

In the first code fragment, the identifier "Firm" is set to the constant value 0x200; this is the base of the firmware vector table. The call uses the index of **qputc** in the firmware vector table (80 decimal) multiplied by 4 (since each pointer is four bytes long). The result is made indirect by the '*'. The second code fragment is useful when a number of references will be made to the same routine. The identifier "qputc" is defined as the absolute address of the pointer to **qputc** in the firmware vector table.

Chapter 3

DMDEBUG

PAGE

Chapter 3

DMDEBUG

A discussion of **dmdebug** features and operation is in the section "DMDEBUG FEATURES." The section "USING DMDEBUG" contains an example of **dmdebug** used on a program. Refer to the table of contents for "pointers" to particular topics.

Dmdebug is a debugger which runs in the **layers** environment on the DMD terminal. It can be used both as a source level symbolic debugger for C language programs and an assembly level debugger for WE*-32000 microprocessor code.

Users communicate with **dmdebug** via using the mouse or typing the commands in **dmdebug**'s layer. The use of the mouse and keyboard is explained in the following sections.

---

* Registered trademark of AT&T Technologies, Inc.

Note: DMD programs using **dmdebug** must be compiled with "**dmdcc -g**".

For example, to compile a program "*file*" for use with **dmdebug**, type:

dmdcc -g *file*.c

DMDEBUG FEATURES

The **dmdebug** features are generally divided into these major categories:

- Layer selection
- Execution control
- Examination and modification of memory and registers
- Context control.

Context control is the programming environment (local variables and registers) for a particular function in the program. **Dmdebug** allows you to request these features via use of the mouse or the keyboard.

Mouse

Users have different menus displayed by pressing mouse button 2 and button 3. If a menu is too large to be displayed, an item called "**more..**" is displayed as part of the menu selection. This item must be selected to see the rest of the menu.

Button 2

The only thing that button 2 permits is displaying memory. Memory can be displayed as bytes, longs (32 bits), shorts (16 bits), structures, and unions. The command ". *expression*" must be typed at the keyboard first before button 2 can be used. This command will display the memory at address "*expression.*" At this point, button 2 may be used to display different memory locations. When button 2 is pressed, a menu appears with the top three items being the location and value for the current memory location (tagged by "**.**") and its two neighbors. Selecting one of these items writes it into the scrolling area and makes that location the current one. If there is a symbolic form for the current location, it is displayed as an item tagged by "**.**". Selecting the current location gives a fuller symbolic form. To exit button 2 mode, click button 1 or 3.

Button 3

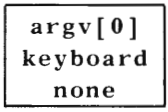
Button 3 items offer features in all four categories of debugging, whereas button 2 items permit access to memory only. At the hit of button 3, you are offered a menu of items to be selected. These items are divided into the major categories below.

Layer Selection

The following items in the menu enable layer selection:

layer
: Selects a new layer to debug. The layer is identified by selecting it with the **target** cursor. When a valid layer has been selected, **dmdebug** asks for an object file with symbol tables.

The menu shows:



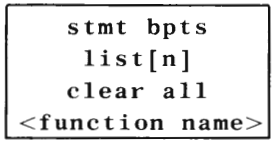
The name of the downloaded file is displayed as **argv**[**0**]. If **argv**[**0**] is the desired file, simply select it using the mouse cursor. To select another file, select **keyboard**, then type the desired file name. Otherwise, select **none** to cancel.

quit

Ends the debugging session for the current layer. This command must be confirmed by hitting button 3 again.

Execution Control

Breakpts displays a menu with the following items:



Selecting **list** lists all active breakpoints. The value **n** in the **list** command indicates the number of breakpoints that are currently set. Selecting **clear all** clears all active breakpoints. As in **quit**, this command requires confirmation. Selecting **stmt bpts** allows you to set a breakpoint on certain C language statements in the program. It displays a menu of file names and line numbers.

Selecting a function offers a further menu with:

One of these items may be tagged by ">" indicating the breakpoints that are currently set. Selecting an item modifies the breakpoints on the function. The breakpoint at function call is after the stack frame has been advanced, but before the initialization of registers and automatics. The breakpoint on function return is after the return value has been set, but before the stack frame is retracted.

Stmt step executes a single C language statement of the program. After the execution, the status line shows the *next* statement to be executed. If the statement executed is a return, the current context will not be set back to the calling function, so **function** or **traceback** must be used explicitly before local variables can be accessed. Statement stepping appears in the menu after stopping at a breakpoint. Single stepping WE-32000 microprocessor instructions can be done by using the keyboard command "**.si .**".

Halt suspends execution of the process.

Go resumes execution of the process.

Memory/Register Examination/Modification

Button 3 allows only memory examination at a high level. Button 2 and the keyboard allow examination of raw memory, any memory modification,

or any register examination. For memory examination in a high level, the following commands can be used:

globals
Repeatedly offers a menu with the program's globals and the system globals. Each one selected is displayed. If a variable selected is a structure, a pointer to a structure or an array, it is displayed:

```
variable.      <field>
variable->     <fields>
variable       <index>
```

Button 3 can be used to complete the expression and display its value.

func_id() vars
Repeatedly offers a menu with the program's local variables, arguments and statics in the current stack frame. The variables are classified as **lcl** (local), **reg** (register), **sta** (static), or **arg** (argument).

Context Control

The following items control context:

traceback
Lists the stack frames from the one currently executing back to **main()**. Each line displayed with the **traceback** command has the form:

```
file: line [+offset] in function(arg,..,arg)
```

giving the statement being executed as source file, line number (and byte offset) within a function and its arguments. If line number information is stripped from the object file, line numbers are not displayed.

function
Offers a menu of functions in traceback order. When one is selected, it becomes the current context to be debugged.

Button 2 Memory Examination

Button 2 permits examination of the following:

byte
Displays a byte at memory locations specified.

short
Displays a short (16 bits) at memory locations specified.

long
Displays a long (32 bits) at memory locations specified. If the value at the current location makes sense as an address, the location referenced and its value are shown tagged by *. This item can be chosen to enable pointer chasing.

decimal
Displays memory as decimal.

octal
Displays memory as octal.

hex
Displays memory as hex.

ascii
Displays memory as ASCII.

string
Displays the contents of the three memory locations as strings. Each string is truncated to be 32 bits long. In other words, this selection displays current_ location - 4, current_location, and current_ location + 4 as strings.

struct
Offers menus of the structures in the program. Selecting **struct** interprets memory as a structure.

enum
Offers menus of the enumerations in the program. Selecting **enum** interprets memory as an enumeration.

Keyboard

Dmdebug also accepts commands at the keyboard. Several of these commands act the same as button 3 and button 2 while the others provide new features.

Layer Selection:

.la
Selects a new layer to debug. The layer is identified by hitting it with the **target** cursor. The communication required is exactly the same as that of using button 3.

control d
Ends a debugging session for the current layer. Requires confirmation on button 3.

Execution Control:

.h
Suspends execution of the process.

.g [*count*]
Resumes execution of the process. If *count* is given, *count* breakpoints are skipped.

.bc function_id
Sets a breakpoint at function call after the stack frame has been advanced, but before the initialization of registers and automatics.

.cc function_id
Clears breakpoint at the function call.

.br function_id
Sets a breakpoint at function return after return value has been set, but before the stack frame is restructured.

.cr function_id
Clears breakpoint at function return.

.bs file line_no
Sets a breakpoint at a given line within a source file.

.cs file line_no
Clears breakpoint at a given line within a source file.

.bl
Lists all breakpoints.

.ca
Clears all breakpoints.

.bi
Sets breakpoint on the WE-32000 microprocessor instruction at the current location.

.ci
The breakpoint at the current location is cleared.

.si [*count*]
Executes the next *count* number of instructions. If *count* is omitted, only one instruction is executed.

.ss [*count*]
Executes the next *count* number of C statements. If *count* is omitted, only one statement is executed.

Memory/Register Examination/Modification:

.b
Displays a byte of memory at the current location.

.sh
Displays a short (16 bits) of memory.

.l
Displays long (32 bits) of memory.

.i
Displays the WE-32000 microprocessor instruction at the current location.

> ***Note:*** For the keyboard commands **.b**,**.sh**,**.l**, and **.i** pressing RETURN 'bumps' the pointer up by the last selected size and displays the location in the previous format.

.v
Prints all local variables in the current function.

struct_id
Displays memory from the current location as an instance of the structure whose identification is **struct_id**.

%< *register name* >
Displays appropriate register. Valid registers names are r0 - r8, fp, ap, psw, sp, pcbp, isp and pc.

*****

Sets the address of the current location to the 32 bit contents at the current location.

.li file line_no

Displays the first instruction of code generated for the statement beginning at a line number of a source file.

.=expression

Sets the current memory location to the value of the expression. Operand length is that of the most recent choice (that is, .b, .w, .l, .sh command from the keyboard, or button 2 equivalent).

.expression

Sets the address of the current memory location to the value of the expression.

Context Control:

.c [*count*]

Sets the new current context to the function which the present one calls. If *count* is given, iterate this process *count* number of times.

.r [*count*]

Sets the new current context to the function which the present one returns. If *count* is given, iterate this process *count* number of times.

.t

Displays a function traceback and sets the current context to the deepest user function.

Miscellaneous:

expression
Evaluates and prints its value. The following operators are supported:

+ - * / -> [ ] . = as binary,

& * - as unary.

Functions may also be called, but are executed by **dmdebug**'s process. Identifiers are bound to variables in the current context according to C's scope rules.

.[*radix*]
Sets desired radix. Valid radices are 8, 10 or 16.

.x
Displays the symbolic name for the current memory location.

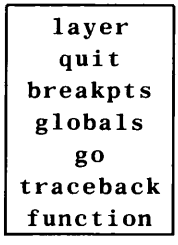
< **file**
Reads command script from a file.

USING DMDEBUG

The program debugger for the DMD is **dmdebug**. **Dmdebug** only runs in **layers**. You choose the layer to debug by pointing with the mouse cursor.

After selecting a layer, the mouse cursor selects locations in memory on mouse button 2, and the top-level menu on mouse button 3.

The top-level menu selections from mouse button 3 are:



Button 3 shows secondary menus if **breakpoints**, **globals**, or **functions** is selected. Further submenus may appear depending on the menu and the item selected. When viewing variables, you can look at the smallest element of an array or structure using these submenus.

These menus describe the process only in terms of objects defined in the source text and assume no knowledge of the implementation. You may examine global variables, halt the process, or set breakpoints at function entry or return. Once halted, you may obtain a traceback, identify an active function and examine its local variables.

If a variable selected for display is a structure, a pointer to a structure, or an array, further menus are presented to choose fields, reference pointers, and index elements, respectively. For example, suppose a variable **p** points into a two-way linked list in which each node is a structure cell with fields **data**, **previous** and **next**, which point to the node's data and its neighbors. Choosing **p** displays its value (an address) and then the expression template:

```
p - > <field>
```

To follow the pointer by completing the expression, you select an item from the menu:

```
cell{} data previous next
```

The items in this menu are the structure name, **cell**{}, and the fields of the structure. Choosing **cell**{} prints all the fields of the structure, producing something like:

```
{data=7516400,previous=0,next=7516360}
```

Choosing next prints the value of the pointer **p->next**. But now, since this is again a pointer to a structure, it is treated in the same manner as **p**. The expression template this time is:

```
p-> next-> <field>
```

but the menu is the same, since the structure is again a cell. This procedure can be carried to an arbitrary depth building bigger and bigger expressions which step through different kinds of structures and arrays. It makes it easy to build expressions that one would not usually attempt to type by hand.

If a structure arrived at is one of the graphics primitives, it can be displayed graphically rather than symbolically and numerically. Points and rectangles are highlighted at their position within the layer in which the debugged process is operating. **Bitmap**s and **Textures** are clipped and copied into the debugger's layer at a position chosen by the user. This is particularly valuable for off-screen **Bitmap**s, where it is virtually impossible to interpret the raw bits.

Refer to **dmdebug** in the *5620 Dot-Mapped Display Reference Manual.*

Example

The following is an example using **dmdebug** on a copy of the demonstration program, **demo clock**. The example assumes that **layers** has already been downloaded to the DMD.

Note: The numbers given in the example may not coincide with the ones you will get. This is because the numbers are addresses that depend on the current state of the DMD.

1. Copy the demonstration program, **clock.c**, into your current directory:

```
cp $DMD/src/demo/layers/clock.c clock.c
```

2. Compile your copy using the **-g** option of **dmdcc**:

```
dmdcc clock.c -g -o clock
```

3. Load **dmdebug** into a new layer. After downloading has finished and initialization (" Initializing..." in the **dmdebug** layer) has completed, the context (" no layer") is displayed in the top part of the layer.

4. Load your recompiled **clock** into another layer:

```
32ld -z clock " `date`"
```

   The -z option of **32ld** will prevent **clock** from executing after it has been downloaded. This allows **dmdebug** to examine the program from the start.

5. Next, tell **dmdebug** which layer you are debugging. Using mouse button 3, select " layer" from the **dmdebug** menu. Use the mouse cursor (now a " target") to select the **clock** layer. Move the cursor over the **clock** layer and click button 3. The **dmdebug** layer displays:

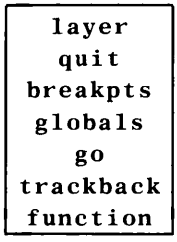
```
argv[0] = clock
symbol tables?
```

6. Move the mouse cursor to the **dmdebug** layer. Press mouse button 3. The menu displays **argv**[**0**], **keyboard**, and **none**. Select **argv**[**0**]. The top portion of the **dmdebug** layer displays:

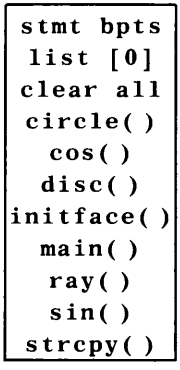
```
halt: pc=7490824
```

   The address **7490824** is the start of the **clock** program.

7. Now, set a breakpoint and start the **clock** program. Press button 3. The menu shows the following:



8. Select the **breakpts** option from the menu, then press button 3. This time the menu shows these options:

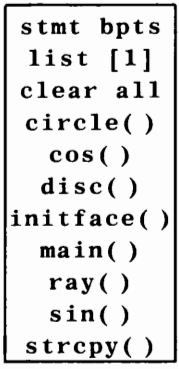


9. You want to execute the program and stop it before it actually executes the first statement. In order to do this, select **main ( )** from the menu.

10. Press button 3. The menu displays these options:

11. Select **call** from the menu.

12. Press button 3. The menu displayed is almost the same as the menu in Step 8, except **list** [ ] contains [**1**]:



13. Select **list** [**1**] from this menu. The screen displays:

```
1: call main( )
```

14. Press button 3. Select **ray( )** from the menu. This breakpoint will be used later to demonstrate the traceback function.

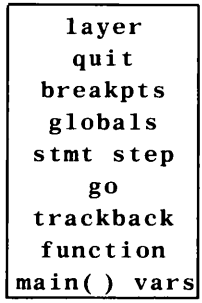
15. Now, return to the main menu. To do this either press button 3 and release it without making a selection from the menu or simply click button 1.

16. Select **go** from the button 3 menu. The top of the layer will momentarily display **running**, and then it will display:

```
call: main(argc=1,argv=7490768) in clock.c:21
```

The **clock** program is now halted at a breakpoint.

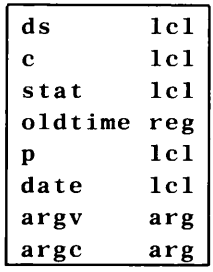
17. Now, examine a few of main's local variables. Examine the menu on button 3:



Notice the two new entries - **stmt step** and **main( ) vars**.

18. Select **main( ) vars** from the menu.

19. Press button 3. The menu displays:



Select **argc** from the menu. The display shows:

```
int: argc=2
```

20. Again, pressing button 3 and using the same menu as before, select **argv**. The following is displayed:

```
**char: argv=7490768
**char: argv ?
```

This states that **argv** is a pointer to a character (that is, a pointer to a string).
The **?** on the second line indicates further refinement can be obtained using button 3.

21. Press button 3, the menu displays:

```
~[ ? ]
for( ? ) ~[ ]
```

The tilde is substituted for the name of the variable; in this case, **argv**.
This menu allows you to examine individual characters of **argv**:

```
~[ ? ]
```

or strings:

```
for( ? ) ~[ ]
```

22. Select the first option.

23. Select the " 1" from the next menu.
The following is displayed in the **dmdebug** layer:

```
*char: argv[1]=7490786
argv[ ? ]
```

24. Pressing button 3 displays a menu that looks something like:

```
~[ ? ]
for( ? ) ~[ ]
"Mon May..."
```

This menu displays individual characters if the first option is

selected; groups of characters when the second option is selected; and the full string if the last option is selected.

25. Select the second option. The display is updated with the following:

```
for($i= ? ;...) argv[1][$i]
```

26. Press button 3 and select " 0" . The last line in the **dmdebug** layer changes to:

```
for($i=0;$i<= ? ;++$i) argv[1][$i]
```

27. Press button 3 and select " 5" . The last line changes and the following is displayed:

```
argv[1][0]='M'
argv[1][1]='o'
argv[1][2]='n'
argv[1][3]=' '
argv[1][4]='M'
argv[1][5]='a'
for($i= ? ;...) argv[1][$i]
```

28. Go back to the top level menu and statement step thru the program. This will demonstrate the use of statement stepping and globals. Press button 1 three times. This will return you to the top level menu. You can also move back up through the menus by using button 3. To do this, press button 3 and release it without selecting any options.

29. Select the **stmt step** option from the button 3 menu. The top part of the **dmdebug** layer contains:

```
stepped: clock.c29  rectf(&display, Drect, F_XOR);
```

30. Statement 29 has not yet been executed. Watch the **clock** layer when you select the **stmt step** option again. The inside portion of the layer is video inverted and the **dmdebug** layer contains:

```
stepped: clock.c30  if(argc!=2){
```

31. Select **stmt step** from the menu. The following is displayed in the **dmdebug** layer:

```
stepped: clock.c36 initface( );
```

32. Select **stmt step** from the menu. The following is displayed in the **dmdebug** layer:

```
stepped: clock.c37 strcpy(date, argv[1]);
```

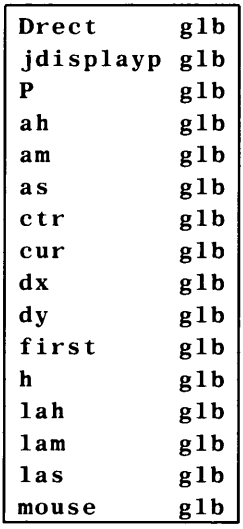
The **clock** layer now has the outline of a **clock** face drawn in it.

33. Select **stmt step** three more times and the following is displayed sequentially in the **dmdebug** layer:

```
stepped: clock.c38 h = atoi2(date+11);
stepped: clock.c39 m = atoi2(date+14);
stepped: clock.c40 s = atoi2(date+17);
```

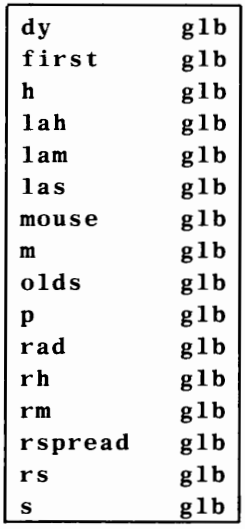
34. Select **globals** from the button 3 menu.

35. Press button 3 again. The menu displays:



36. Notice this menu has a scroll bar associated with it. While still pressing button 3, position the cursor in the scroll bar area and move the cursor down the screen. Stop when the scroll bar is positioned at the end of the menu.

The menu displays:



37. Select the **s glb** menu entry. The **dmdebug** layer shows the following:

    int: s=<*nn*>

    where <*nn*> is the number of seconds.

38. Now let's take a look at the traceback function. We will use the **ray( )** breakpoint set earlier. Select the the **go** menu item with button 3. The top part of the **dmdebug** layer will temporarily display:

    running

    This will be replaced by:

    call: ray(r=106,ang=258,rspr=5) in clock.c:131

    which tells us the **clock** routine was halted just before statement 131 in **clock.c**. The numbers within the parentheses will differ each time this example is followed.

39. Select **traceback** from the button 3 menu. The lower half of the **dmdebug** layer contains:

```
task.c in sw(run=0)
clock.c131+21 in ray(r=106,ang=258,rspr=5)
clock.c:86 in main(argc=2,argv=7472984)
```

This tells us the program is stopped at the **sw( )** routine. This routine was called by **ray( )** which was called by **main( )**.

40. As a last example let's try some pointer chasing. The pointers we will be using are related to the DMD's control of the layer. There will be no attempt to explain what these pointers are used for. Select **globals** from the button 3 menu.

41. Press button 3. Select **P** from this menu:

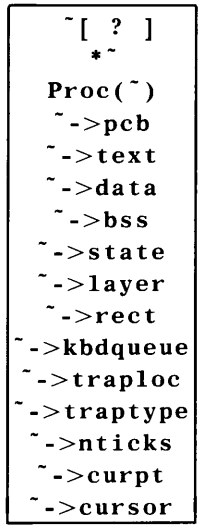
```
Drect      glb
jdisplayp  glb
P          glb
ah         glb
am         glb
as         glb
ctr        glb
cur        glb
dx         glb
dy         glb
first      glb
h          glb
lah        glb
lam        glb
las        glb
mouse      glb
```

The **dmdebug** layer displays:

```
*struct Proc: P=7462664
```

which states that **P** is a structure of type **Proc**.

42. Press button 3. The following menu is displayed:

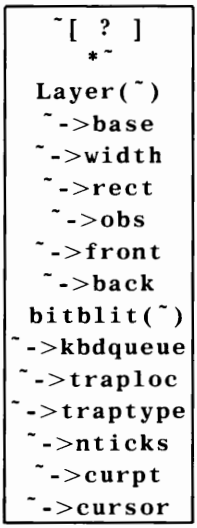


The tilde character is an abreviation for **P**. The -> characters are standard C language syntax for "pointer to".

43. Select the ~->**layer** item from this menu. Now the **dmdebug** layer shows the following:

```
*struct Layer: P->layer=7472880
*struct Layer: P->layer ?
```

and the menu on button 3 shows:



More detail on **dmdebug** appears in the *5620 Dot-Mapped Display Reference Manual.*

Chapter 4

PROGRAMMING LAYERS

PAGE

Chapter 4

PROGRAMMING LAYERS

The tutorials for the "**-f**" and the "**hagent**" features of **layers** are contained in the sections "THE LAYERS -F OPTION" and "THE LAYERS HAGENT," respectively. Examples of using these features are in the sections "Layers -f Example" and "Layers Hagent Example."

A particular **layers** configuration can be defined from a shell script or C language program using the **layers -f** or **hagent** feature. You can initialize and define a particular **layers** setup and load programs into a specified layer.

For example, one user may need a text development setup, another may need a graphics development setup, another may need a combination of both, and so on. The user can "call up" these configurations by typing one command. The DMD will set up each layer on the display, then load and execute the desired program in the appropriate layer.

THE LAYERS -F OPTION

The **layers -f** option will automatically produce a layer configuration, load, then execute programs in the layer. This allows you to create a **layers** setup for any application. For example, you could design a text processing setup with one layer running the DMD editor **jim**, another layer running a UNIX System shell, another running the UNIX System editor **vi**, and so on. A program development setup may run the DMD compiler (**dmdcc**), the DMD debugger (**dmdebug**), and the DMD editor (**jim**) in different layers on the display.

Using the -f Option

The **layers -f** command syntax is: **layers -f** *file* where *file* contains the coordinates and commands for the layer configuration in the format:

```
origin.x origin.y corner.x corner.y command
```

for each layer in the setup.

The variables *origin* and *corner* define diagonally opposite corners of the layer (and thus the entire layer). *Command* can be either one or more UNIX System commands. If more than one command is used, they must be separated by semicolons.

> ***Note:*** If an error is detected in a line of the configuration file (*file* above), the line in error and the rest of the configuration file is ignored.

The **-f** option works on screen coordinates: from (8,8), upper left corner of the display, to (792,1016), lower right corner of the display. Layers created with the **-f** option may cover other layers as with normal **layers** operation.

Note: The minimum size of a layer is: 32 dots (x difference) by 32 dots (y difference).

Layers -f Example

This discussion gives an example of the **-f** option. The setup consists of 4 layers with different programs running in each, making up a simple "text development setup." The **-f** option requires a file containing the layer coordinates and the command. The following text is in a file called **example** which will create the simple "text development setup." Each line in **example** has the syntax:

origin.x origin.y corner.x corner.y command; command

where *origin.x* and *origin.y* define one corner of the layer, and *corner.x* and *corner.y* define the diagonally opposite corner. The *command* is the program loaded into the layer and executed.

The file **example** contains:

```
8 8 620 650 jim; exec sh
8 652 700 792 ls; exec sh
8 622 400 1000 proof; exec sh
410 720 790 1010 exec sh
```

The "text development setup" **example** is created by typing:

```
layers -f example
```

followed by RETURN.

The first line in **example** will create a layer with screen coordinates (8,8),(620,650) and load the DMD editor **jim** in that layer. Each layer is created in the same manner, with the first set of coordinates defining the upper left-hand corner and the second set defining the lower right-hand corner of the layer. The command after the coordinate pair is executed in the layer.

When the loading **example** is complete, the DMD display will contain four layers: one with the DMD editor **jim**, one with a list of the files in the current directory, one with the phototypesetter simulator **proof**, and one running the UNIX System shell.

> ***Note:*** Each command line must end with **exec sh**, or the layer will "lock up" when the current program exits.

Up to six layers may be created, overlapping as required. After loading, the DMD operation is normal. Layers can be deleted, reshaped, moved, or created. The **layers -f** option gives the DMD programmer an easy means to initially define an application display.

THE LAYERS HAGENT

The Hagent feature allows you to control **layers** from a custom-designed C language program using **hagent** commands. The **hagent** commands give you all of the functions that appear on the **layers** menu, and automatic loading and execution of DMD programs.

The **hagent** commands operate on "channels." Each channel number corresponds to a layer (numbered 2 through 7 inclusive). The following list contains the **hagent** commands with a short description of each.

COMMAND **DESCRIPTION**

openagent( )

opens the DMD control channel. Upon successful completion, **openagent** returns a file descriptor that is used as input to all routines except **openchan** and **Runlayer**. Otherwise, the value EOF is returned.

New(cntlfd, origin.x, origin.y, corner.x, corner.y)

creates a new layer with a separate shell. **Cntlfd** is the file descriptor obtained from an **openagent** and {**origin.x, origin.y, corner.x, corner.y**} are the coordinates of the layer rectangle. The layer appears on top of any overlapping layers. The layer is NOT made current (the keyboard is not attached to the new layer). Upon successful completion, **New** returns the DMD channel associated with the layer. Otherwise, the value EOF is returned.

> ***Note:*** If all the coordinates arguments are zero, the command modifies the arrow cursor to the 'sweepcursor,' allowing the user to sweep a layer by pressing and holding mouse button 3.

Runlayer(chan, command)

runs **command** in the layer associated with channel **chan**. The layer must be created with **New**.

Current(cntlfd, chan)

makes the layer associated with channel **chan** current (that is, the layer is attached to the keyboard). **Cntlfd** is the file descriptor obtained from an **openagent**.

COMMAND **DESCRIPTION**

Delete(cntlfd, chan)

deletes the layer associated with channel **chan** and kills all host processes associated with the layer. **Cntlfd** is the file descriptor obtained from an **openagent.**

Top(cntlfd, chan)

makes the layer associated with channel **chan** appear on top of all overlapping layers. **Cntlfd** is the file descriptor obtained from an **openagent**.

Bottom(cntlfd, chan)

puts the layer associated with channel **chan** under all overlapping layers. **Cntlfd** is the file descriptor obtained from an **openagent**.

Move(cntlfd, chan, origin.x, origin.y)

moves the layer associated with channel **chan** from its current screen location to a new screen location at the origin point **(origin.x, origin.y)**. The size and the contents of the layer are maintained. **Cntlfd** is the file descriptor obtained from an **openagent**.

Reshape(cntlfd, chan, origin.x, origin.y, corner.x, corner.y)

reshapes the layer associated with channel **chan**. **Cntlfd** is the file descriptor obtained from an **openagent** and {**origin.x, origin.y, corner.x, corner.y**} are the new coordinates of the layer rectangle. The contents of the layer will not be maintained.

Note: If all the coordinates arguments are zero, the command modifies the arrow cursor to a 'sweepcursor', allowing the user to sweep a layer by pressing and holding mouse button 3.

COMMAND **DESCRIPTION**

Exit(cntlfd)

exits the **layers** program and kills all processes associated with **layers. Cntlfd** is the file descriptor obtained from an **openagent**.

Newlayer(cntlfd, chan, origin.x, origin.y, corner.x, corner.y)

creates a new layer. **Cntlfd** is the file descriptor obtained from an **openagent** and {**origin.x, origin.y, corner.x, corner.y**} are the coordinates of the layer rectangle. The layer appears on top of any overlapping layers. The layer is NOT made current (the keyboard is not attached to the new layer). Upon successful completion, **Newlayer** returns the DMD channel associated with the layer. Otherwise, the value EOF is returned.

> ***Note:*** If all the coordinates arguments are zero, the command modifies the arrow cursor to a 'sweepcursor', allowing the user to sweep a layer by pressing and holding mouse button 3.

openchan(chan)

opens the channel, **chan**, obtained from a **New** or **Newlayer.** Upon successful completion, **openchan** returns a file descriptor that can be used as input to **write(2)**. Otherwise, the value EOF is returned.

Caution: By using these commands you can create and destroy layers and the channels that connect them to the UNIX System. If you try downloading into a layer already running another program, the results are indeterminate and potentially unpleasant.

Using Hagent

Programs compiled with hagent commands require the **hostagent.o** file. To compile an hagent program, type:

cc *file.c* **$DMD/lib/hostagent.o**

where *file.c* is the hagent program source.

Programs containing hagent commands can be executed only in **layers**. This requires loading **layers**, creating a layer, then executing the hagent program in the layer.

For example, to execute an hagent program, follow these steps:

1. Edit and compile the hagent program. To compile, type: **cc** *file.c* **$DMD/lib/hostagent.o**
2. Load **layers**.
3. Create a layer.
4. Load and execute the hagent program. Type: **a.out**

You may want to simplify this procedure for users by creating a **layers -f** file that creates a layer, then executes the hagent program. The procedure could be reduced to executing **layers -f** from a programmable function key or from a user's ".profile" at login.

Layers Hagent Sample

The following C language example uses **hagent** commands to create two layers, load and execute **demo clock** in the second layer, bring the first layer to the top, then make it current.

```
#include   <stdio.h>
#include   <sys/jioctl.h>

main( )
{
 int    cntlfd;
 int    chan1, chan2;

 if ( (cntlfd = openagent( )) == EOF )
  {
   printf( ''openagent error\n'');
   exit (0);
  }

/* create new layer with shell
   at {100, 100, 400, 400}  */

 if ( (chan1 = New(cntlfd, 100, 100,
                400, 400)) == EOF )
       perror(''error from New'');

/* create new layer with shell
     at {0, 0, 150, 150}
   and run ''demo clock'' in it */

 if ( (chan2 = New(cntlfd, 0, 0, 150,
                150)) == EOF )
       perror(''error from New'');
 if ( Runlayer(chan2, ''demo clock'') == EOF )
       perror(''error from Runlayer'');

/* put {100, 100, 400, 400} layer on
   top and make it current */

 if ( Top(cntlfd, chan1) )
      printf(''error from Top\n'');
 if ( Current(cntlfd, chan1) )
      printf(''error from Current\n'');

return;
}
```

Chapter 5

ICON

PAGE

Chapter 5

ICON

The icon and texture editor for the DMD is **icon**. With **icon**, you can create small textures that can be used in your DMD program to show what activity is taking place in your program.

To load and execute **icon**, see the section "USING **icon**." **icon** commands are described in the section "THE **icon** MENU." Detailed descriptions of **icon** functions follow in the section "DRAWING WITH **icon**." The section "INCLUDING ICONS AND CURSORS IN PROGRAMS" describes the steps to put your icons and cursors into DMD programs.

Icon Editor Definitions

For the following discussion on the DMD **icon** editor, a few definitions are necessary:

icon	Is the icon and texture editor for the DMD. It is shown in lower case, bold type.
texture	Is a pattern drawn in the **icon** grid, before defining it as an icon or Texture16.
Texture16	Is a 16 by 16 pixel drawing in the **icon** grid, defined with a mouse button 2 press. Texture16s are stored as a 16 by 16 bitmap.
icon	A drawing of any size in the **icon** grid, defined with a mouse button 3 press. Icons are stored as the data portion of a **Bitmap**.

See **ICON(1)** in the *5620 Dot-Mapped Display Reference Manual* for more information.

ICONS AND TEXTURE16s

You can create two types of textures with the **icon** editor: an "icon" or a "Texture16." An icon has *no* size restriction and many smaller icons may be combined to build a larger icon. The **icon** menu is a good example of using icons to select the various **icon** commands. Do not use icons as mouse cursors. The processing overhead to update (move) the large "icon cursor" is not efficient nor necessary. Mouse cursors are created with Texture16s.

A Texture16 is a 16 by 16 bitmap which can be used by your program to direct user activity and give status information (such as mouse cursors). For example, the **layers** program uses Texture16s to draw the arrow, target, mouse, and coffeecup cursors.

Read and Write Textures

The **icon** program reads, writes, and changes both icons and Texture16s. You define which type of texture by pressing either mouse button 2 (for a Texture16) or button 3 (for an icon) after selecting a write, read, or change command. For example, when you have drawn a texture and select the write command, **icon** requests which type of texture you want by changing the mouse cursor to the "sweep rectangle" cursor. Pressing button 2 will define the texture as a Texture16. Pressing button 3 will define the texture as an icon.

How Textures Are Stored

The icon and Texture16 files saved by **icon** are stored differently. An icon file contains a list of 32-bit quantities which correspond to the data portion of an icon bitmap. The file does not include the header and trailer information necessary to use it in a C language program. This allows you to build larger icons. You must include the header and trailer information to use the icon in your program.

The Texture16 file is saved with the appropriate header and trailer information to include in your program. You do not have to add any additional information. The following sections give examples on how to draw and use textures in your program.

Define Icons and Texture16s

Many **icon** operations require the texture in the grid be defined as either an icon (a bitmap of any size), or a Texture16 (a 16 by 16 bitmap). When the "sweep rectangle" cursor appears, **icon** is requesting that you define the texture. Pressing mouse button 2 will define the texture as a Texture16. Pressing mouse button 3 will define the texture as an icon.

If you press button 2, the cursor changes to a "large box" that covers a 16 by 16 pixel area in the grid. Move the cursor over the section of the grid you want defined, then release button 2. The texture contained in the "large box" is defined as a Texture16.

If you press button 3, the cursor changes to the "sweep cursor" icon. To select an area in the grid, press and hold button 3 to fix one corner of the rectangle. Sweep the rectangle to enclose the desired area on the grid. Release button 3. The texture contained in this rectangle is defined as an icon.

USING ICON

Load the Icon Editor

Create a layer. Load **icon** into that layer. The default display consists of a large grid representing 50 by 50 pixels in the lower right-hand corner of the layer. Each square in the grid represents one pixel on the DMD display.

The default size of the grid may be changed by using the "**-x**" and "**-y**" options. For example, to display a 42 by 34 pixel grid, type:

```
icon -x 42 -y 34
```

While the cursor is positioned over the grid, and the **icon** layer is current, pressing mouse button 1 draws a pixel, and pressing button 2 undraws a pixel. In the upper left-hand corner of the display, the texture you are drawing in the grid will appear actual size. Figure 5-1 shows the **icon** layer with the supplied icon "banana" drawn in the grid.

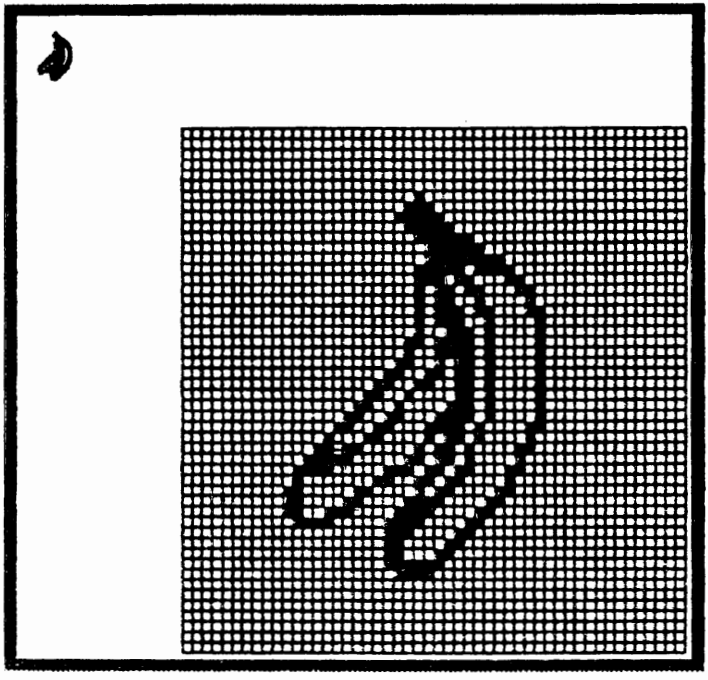


Figure 5-1. The Icon Editor Display

When changing your texture, **icon** requires you to define it as a Texture16 (button 2) or as an icon (button 3). After you make a menu selection, the cursor changes to the "sweep rectangle" cursor. Define the texture as a Texture16 by pressing button 2, or as an icon by pressing button 3.

The **icon** help message is displayed by selecting the "help" icon in the **icon** menu.

THE ICON MENU

Pressing button 3 raises a menu consisting of commands represented graphically. For example, a "quill pen" icon represents the write command. Figure 5-2 shows the **icon** menu.

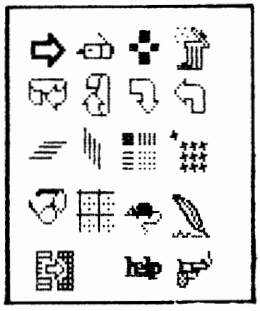


Figure 5-2. The Icon menu

There are nineteen commands, one of which is "help," which brings up the **icon** help message.

To select an **icon** command, press and hold button 3. Move the mouse cursor (now a box) over the desired icon. Releasing button 3 will select the command. For example, to select the "help" command:

1. Press and hold mouse button 3.
2. Move the mouse cursor over the "help" icon; release button 3.
3. Every icon that appears in the menu is shown in the "help" message with a short description.
4. Click button 3 to return to drawing on the grid.

The following text gives a short description of each command in order from left to right, top to bottom (as shown in the menu).

MENU	DESCRIPTION
	**Arrow**: Moves the icon to another portion of the grid.
	**Copier**: Copies the icon to another portion of the grid.
	**Invert**: Inverts video on a texture.
	**Erase**: Erases the texture.
	**Reflect X**: Flips the texture on the y axis.
	**Reflect Y**: Flips the texture on the x axis.
	**Rotate +**: Rotates the texture clockwise.
	**Rotate –**: Rotates the texture counterclockwise.

MENU	DESCRIPTION
	**Shear X**: Shears the texture along the x axis. (For example, a rectangle will change to a parallelogram shifted along the x axis.)
	**Shear Y**: Shears the texture along the y axis. (For example, a rectangle will change to a parallelogram shifted along the y axis.)
	**Stretch**: Stretches (or shrinks) the texture.
	**Duplicate**: Duplicates a texture over a larger (or smaller) area of the grid.
	**Read File**: Reads an icon (or Texture16) file.
	**Background Grid**: Draws a reference grid divided into 16 by 16 squares with borders highlighted.
	**Pick Cursor Icon**: Changes the mouse cursor to the Texture16 displayed on the grid.

MENU	DESCRIPTION
	**Write File**: Writes an icon or Texture16 file.
	**Bitblt**: The bitblt operator. Allows you to alter the source and destination areas specified on the grid.
	**Press a Button to Continue**: Prints the **icon** help message.
	**Exit**: Exits the **icon** program. Confirm request with a button 3 press.

The following icons appear as mouse cursors after selecting one of the icons from the menu.

ICON	DESCRIPTION
	**Wait**: Wait while **icon** requests I/O.
	**Mouse Inactive**: The mouse is inactive, wait.
	**Menu on Button 3**: Press button 3 to display menu.

ICON	DESCRIPTION
	**Sweep Rect**: Sweep a rectangle (button 3). The rectangle defines the size of an icon.
	**Sweep Rect or Get 16 x 16 Frame**: Sweep a rectangle (button 2 or 3). Button 3 defines an icon. Button 2 defines a Texture16.

DRAWING WITH ICON

Generally, you can create textures by pointing to the **icon** grid with the mouse cursor. The bits pointed to by the mouse cursor are turned on (reverse video) by clicking button 1, and off (normal video) by clicking button 2. You may draw (or undraw) a bit at a time by clicking the button; or, by holding the button depressed, draw (or undraw) many bits while sweeping the cursor.

Move Texture

A texture is moved within the **icon** grid by using the "arrow" menu selection as follows:

1. Select the "arrow" command by pressing mouse button 3. Move the "box" cursor over the "arrow" in the upper left-hand corner of the **icon** menu.
2. Release button 3.
3. Define the texture as an icon or Texture16.

4. Move the mouse cursor over the area of the grid where you want the icon (or Texture16) to appear; then click button 3.

5. The icon (or Texture16) is drawn into the new location on the grid.

Copy Texture

After you have drawn your texture, you may copy it to another portion of the grid using the **icon** "copy" command. The "copy" command is selected as follows:

1. Press button 3 to display the **icon** menu.

2. Place the mouse cursor over the "copier" icon.

3. Release button 3.

4. When you select the **icon** copy command, the mouse cursor changes to the "sweep rectangle" cursor. Pressing button 2 defines the texture in the grid as a Texture16. Pressing button 3 defines the texture as an icon (see "Defining Icons and Texture16s").

5. Move the cursor to the portion of the grid where you want the copy to appear.

6. Click button 3.

7. The copy appears in the desired area. Note that copies (and the original icons) are limited to the grid area.

Note: You can copy the icon as many times as you wish. However, the texture may become garbled and unrecognizable as copies overlap. Watch the actual size representation of the texture in the upper left-hand corner of the **icon** layer. This is a true indication of the texture in the grid.

Invert Video

To invert the video of a section of the **icon** grid:

1. Press mouse button 3.
2. Move the cursor over the "invert" command (top row, third from the left). Release button 3.
3. Define the texture as an icon or Texture16.
4. When button 2 (or 3) is released, the area covered by the cursor is inverted.

Erase Texture

To clear an area of the **icon** grid:

1. Select the "erase" command from the menu.
2. Define the texture as an icon or Texture16.
3. When button 2 (or 3) is released, the area covered by the cursor is deleted.

Flip Texture

The "Reflect X", "Reflect Y", "Rotate –", and "Rotate +" commands flip the picture in the indicated direction. The procedure is the same for these operations:

1. Select the desired command from the **icon** menu.
2. Define the texture as an icon or Texture16.
3. At the release of mouse button 2 or button 3, the texture flips.

Shear Texture

The "Shear X" command shears the texture along its horizontal axis. To shear the texture, first define it as an icon. The cursor changes to a small "box." Move the small "box" left or right. Click button 3. The "Shear X" command causes the y-axis to tilt in the direction of the small "box." The texture 'tilts', or 'skews', in the direction and distance selected by the small "box."

The "Shear Y" command shears the texture along its vertical axis. To shear the texture, first define it as an icon. The cursor changes to a small "box." Move the small "box" up or down. Click button 3. The "Shear Y" command causes the x-axis to tilt in the direction of the small "box." The texture 'tilts', or 'skews', in the direction and distance selected by the small "box."

Change Texture Size

The "stretch" command stretches (or shrinks) the texture. Use this command to expand (or shrink) the texture. After defining the texture as an icon or Texture16, the cursor changes to a "box" cursor. Fix one corner of the new texture by pressing button 3. Move the cursor to the diagonally opposite corner of the new texture. Release button 3. The texture changes to fill the box.

Duplicate Texture

The Texture16 command duplicates a texture over a larger (or smaller) area within the grid. To duplicate a texture:

1. Select the "duplicate" command from the menu.
2. Define the area to be duplicated as either a Texture16 or an icon.
3. Using either button 2 or button 3, define the area where the texture is to be duplicated as a Texture16 or an icon.

4. The texture is repeatedly duplicated until it fills the new area.

Read Icon File

To read an existing icon (or Texture16) into **icon**, select the "read file" command from the **icon** menu. You are required to type in the name of the icon (or Texture16).

If the file is a Texture16, the mouse cursor changes to a 16 by 16 pixel box, defining the area the Texture16 covers on the grid. Move the cursor to the position on the grid where you want the Texture16 to appear. Click button 2. The Texture16 is drawn in the grid.

If the file is an icon, the cursor changes to a box enclosing the area of the grid covered by the icon. Move the cursor to the position where you want the icon to appear. Click button 3. The icon is drawn in the grid.

Background Grid

The **icon** grid can be divided into 16 by 16 pixel areas by selecting the "background grid" icon from the menu. Selecting the "background grid" icon again removes it. These 16 by 16 areas are useful when drawing Texture16s, or to determine how many 16 by 16 pixel areas a texture occupies.

Change Mouse Cursor

You can change the mouse cursor to display the Texture16 you are developing by selecting the "mouse" icon in the menu. The cursor changes to a 16 by 16 pixel box. Move the box over the area you want, then click button 2. The cursor changes to the area you selected. This lets you test the Texture16 for its suitability as a mouse cursor. Clicking any mouse button cancels the command and returns the cursor to its previous shape.

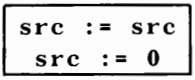
Bitblt Operator

The "bitblt operator" is used to alter the source and destination areas specified on the grid. You can perform bitwise copies and moves on textures in the grid.

To perform a bitwise move:

1. Select the "bitblt operator" command from the menu. The cursor changes to the "dark square in stack" icon.

2. Press and hold button 3. A menu is displayed showing:

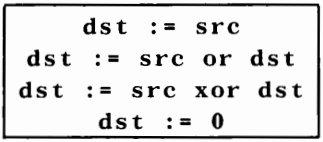


   Where:**src** := **src** copies textures and **src** := **0** moves textures.

3. Select **src** := **src** or **src** := **0**.

4. The cursor changes to "dark square in stack."

5. Press and hold button 3. A menu is displayed showing:



Where:

dst := src
Copies (or moves) the selected texture to another portion of the grid. If the new texture overlaps the original texture, the original is erased.

dst := src or dst
Copies (or moves) the selected texture to another portion of the grid. If the new texture overlaps the original texture, the bits are or'ed.

dst := src xor dst
Copies (or moves) the original texture to another portion of the grid. If the new texture overlaps the original texture, the bits are exclusive or'ed.

dst := 0
Erases the texture.

6. Select an entry from the menu. The mouse cursor changes to the "sweep rectangle" cursor.

7. Define the texture as an icon or Texture16.

8. Move the cursor to the area in the grid you want the texture to appear.

9. Define the texture as an icon or Texture16.

10. The texture is moved (or copied) to the new location in the grid defined by the menu selection.

Quit Icon

To exit the **icon** editor:

1. Select the "exit" command from the menu. The cursor changes to a "smoking gun."

2. Click button 3 to exit the **icon** editor. Click button 1 or 2 to cancel the exit request.

SAVING ICONS

Write Icon files

To write a texture from the **icon** grid to a file:

1. Select the "write file" command from the menu.

2. Define the texture as a Texture16 (button 2) or an icon (button 3).

3. Type in a name for the file. While writing the file, the **icon** layer changes to reverse video.

Icons and Texture16s are saved in the layer's current directory unless a full path name is specified.

INCLUDING ICONS AND CURSORS IN PROGRAMS

Icons

The textures you draw on the grid can be saved in files as icons or Texture16s for use as mouse cursors. Icons are static displays normally used for graphic representations of commands, as with the **icon** menu. You can combine icons to create a larger icon. Note that icons are unsuitable for use as mouse cursors because of their size.

Include Icons in Programs

The icon is saved in a file as the data portion of a bitmap. In your program, the icon is drawn onto the display by the **bitblt** operator that moves the icon (as a bitmap) to layer (or screen) coordinates. A bitmap is a list of 32-bit quantities identified by a specific header and trailer. To allow you to build larger icons, the header and trailer are not put into the file by **icon**. This requires that you include the header and trailer when writing an application program.

The output of **icon** is put into a file with the following format:

```
0xFF008800,
0xFF008800,
0xFF008800,
0xFF008800,
    .
    .
    .
```

The file must be edited to include the header and trailer to declare the icon as an array of data. The file format should appear:

```
Word bits[ ]={
        0xFF008800,
        0xFF008800,
        0xFF008800,
        0xFF008800,
             .
             .
             .
        };
```

Using the output from **icon**, a bitmap is constructed and used in a program, as in the sample program on the next page.

```
#include <dmd.h>
Word bits[6] = {     /*  Header: Defines the start
                         and length of the bitmap.
                         NOTE: 6 denotes the number
                         of 32-bit quantities  */

        0xFF00FF00,  /* start of file */
        0xFF00FF00,  /* saved by icon */
        0xFF00FF00,
        0xFF00FF00,
        0xFF00FF00,  /* end of file */
        0xFF00FF00   /* saved by icon */

        };           /* trailer: defines the */
                     /* end of the icon bitmap */

        /*  beginning of sample program  */

Bitmap example = {
        bits,           /* pointer to data  */
        1,           /* width in words of total
                        data area  */
        0,0,32,6,    /* rectangle  */
        0};          /* unused, always zero  */

main( )                  /* program that draws */
{                    /* the icon bitmap    */
        int i,j;

        for (i=10; i<800; i+=40)
        {
           sleep(20);
           bitblt(&example,example.rect,
           &display,Pt(i,i),F_STORE);
        }

}       /* end of program */
```

The header and trailer information has been added to declare the icon as an array of data. Next, the bitmap must be declared and initialized.

In the sample program, the first value is the name of the data array pointer "bits." The second value is the width of the data in words. To

determine the width of the data, assume the bitmap rectangle begins at the origin (0,0). The width is equal to the number of hexadecimal words (separated by commas) per line in the **icon** output file.

For example,

```
0x12345678, 0x12345678, 0x12345678 \n
```

has a width of three words.

The next four integers initialize the bitmap rectangle corresponding to the original icon. The first two are zeros. In the sample program, the original icon is 32 bits wide by 6 bits long, so the bitmap rectangle is:

```
0,0,32,6.
```

Note the **icon** output file is padded with zeros to 32-bit words. The last integer in a bitmap is unused, and always set to zero.

Cursors

Cursors are Texture16s. An icon *or* a Texture16 may be used as an "icon" as in the **icon** menu, but only a Texture16 may be used as a mouse cursor.

The following is an example of the file **mouse** (the mouse cursor) written by **icon** using mouse button 2. The output is a Texture16.

```
Texture16 mouse = {
        0x0000, 0x0000, 0x03E0, 0x17F0,
        0x3FF0, 0x5FFE, 0xFFF1, 0x0421,
        0x0002, 0x00FC, 0x0100, 0x0080,
        0x0040, 0x0080, 0x0000, 0x0000
};
```

In a DMD program, the cursor is drawn onto the display by the **cursswitch** function which changes the cursor at layer (or screen) coordinates defined

by **mouse**. Refer to the *5620 Dot-Mapped Display Reference Manual* for further information on **cursswitch**.

To use the cursor in your program, note the following:

- Include **request(MOUSE)** to indicate that the mouse resource is requested by your program.
- Use **cursswitch(&** *cursor***)** to switch cursors, where *cursor* is the name of the cursor.
- To switch to the default **layers** arrow cursor, use **cursswitch((Texture16 *)0)**

Include Cursors in Programs

The textures you create with **icon** are easily placed into your programs. In the following example, we will change the 'LET GO' cursor in the demonstration program **doodle** to one designed with **icon**.

The program **doodle** scribbles in a layer while mouse button 1 is pressed. Pressing button 2 brings up a 'LET GO' cursor, and clears the layer. To change the 'LET GO' cursor:

1. Create two layers.
2. Copy **$DMD/src/demo/layers/doodle.c** into your current directory.
3. Load **icon** into one layer.
4. Edit your copy of **doodle.c** in the other layer. Note: Do not edit the original source code; edit your copy only.
5. Using **icon**, create a Texture16 for use as a cursor.

6. Define the texture as a Texture16; then save it with the name 'mesg'.

7. In the layer containing your copy of **doodle.c**, remove the Texture16 'mesg' (6 lines).

8. Read the Texture16 you created with **icon** into your copy of **doodle.c** after #**include** <**dmd.h**>.

9. Be sure the name of your (new) Texture16 is 'mesg', or change all references to 'mesg' to the new name of your Texture16.

10. Write your copy of **doodle.c**. Quit the editor.

11. Compile your copy of **doodle.c** using **dmdcc**. For example, type:

```
dmdcc doodle.c -odoodle
```

    Note: The '**-o**' option places the executable binary in the file **doodle** instead of "**dmda.out**".

12. Load your "new" **doodle** into the layer. For example, type:

```
32ld doodle
```

13. Once your "new" **doodle** is loaded, pressing mouse button 1 lets you scribble in the layer, and pressing button 2 displays your new cursor and clears the layer.

SUPPLIED ICONS

Many icons (and Texture16s) are supplied with this package. They may be used "as is" in your programs or modified to suit the application. They are located under **$DMD/icon** in the following sub-directories:

- ***16x16*** - Texture16s useful for mouse cursors and menus are in **$DMD/icon/16x16**. To display these, change directory to **$DMD/icon/16x16**, then type: **CATALOG *.** The layer will fill with Texture16s. Press RETURN to display more Texture16s.

- ***texture16*** - Texture16s useful as background textures are in **$DMD/icon/texture16.** To display these, change directory to **$DMD/icon/texture16**, then type: **INSPECT** *icon-name*, where *icon-name* is the file name of the icon.

- ***large*** - Large sized icons are in **$DMD/icon/large**. To display these, change directory to **$DMD/icon/large** then type: **CATALOG *.** The layer fills with icons. Press RETURN to display more icons.

- ***face48*** - Various faces are contained in **$DMD/icon/face48**. Faces are reviewed by changing directory to **$DMD/icon/face48**, then reading the face file into the **icon** editor using the "read file" command.

All textures may be read into **icon** by typing the name of the icon (or Texture16). The path to the directory is not necessary as **icon** searches both the current directory and **$DMD/icon** (and its sub-directories) for the named file. These files are read into **icon** by selecting the "read file" command, then typing the file name.

Chapter 6

THE DMD COMPILER

PAGE

Chapter 6

THE DMD COMPILER

This chapter describes the DMD C language compiler, **dmdcc**, and the C programming language the compiler translates. The compiler is part of the WE-32000 microprocessor Software Generation System (SGS).

The SGS is a package of tools used to create and test programs for the WE-32000 microprocessor. These tools allow high-level program coding and source-level testing of code. The C language is implemented for high-level programming; it contains many control and structuring facilities that greatly simplify the task of algorithm construction. Within the SGS, a C language compiler converts C language programs into assembly language programs that are ultimately translated into object files by the assembler, **m32as**. The link editor, **m32ld**, collects and merges object files into executable loads. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at the C-language source level. In addition, a utility package aids in testing and debugging.

THE DMD COMPILER

The main command of the SGS is **dmdcc**; it operates much like the UNIX System **cc** command. To use the compiler, first create a file (typically by using any UNIX System text editor) containing C language source code. The name of the file created must have a special format; the last two characters of the file name must be "*.c*" — as in, *file.c*.

Next, enter the SGS command

dmdcc *options file.c*

to invoke the compiler on the C language source file *file.c* with the appropriate *options* selected from Figure 6-1. The compilation process creates an absolute binary file named **dmda.out** that reflects the contents of *file.c* and any referenced library routines. The resulting binary file, **dmda.out**, can then be executed on the target system.

Options can control the steps in the compilation process. When none of the controlling options are used, and only one file is named, **dmdcc** automatically calls the assembler, **m32as**, and the link editor, **m32ld**, thus resulting in an executable file, named **dmda.out**. If more than one file is named in a command,

dmdcc -c *file1.c file2.c file3.c*

then the output will be placed in files *file1.o*, *file2.o*, *file3.o*, and the executable file, **dmda.out**. The "**.o**" files can be linked and executed through the **m32ld** command.

The **dmdcc** compiler also accepts input file names with the last two characters **.s**. The **.s** signifies a source file in assembly language. If the *-c* option is specified, the **dmdcc** compiler passes this type of file directly to **m32as**, which assembles the file and places the output on a file of the same name with **.o** substituted for **.s**. Otherwise, **dmdcc** calls **m32ld** and creates **dmda.out**.

Dmdcc is based on a portable C language compiler and translates C language source files into Instruction Set 25 (IS25) assembly code. Whenever the command **dmdcc** is used, the standard C language preprocessor is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **dmdcc** and need not be called directly by the programmer. The expanded files are translated from C language to IS25. Then, unless the appropriate flags are set, **dmdcc** calls the assembler and the link editor to produce an executable file.

Instruction Set 25

Instruction Set 25 (IS25) is the assembly language and instruction set for AT&T 3B Computers and WE-32000 Microprocessor Products. IS25 is common to all models. It consists of a complete description (syntax and semantics) of the assembly language, common instructions, and the object level description of data. It is designed to be a space and time efficient instruction set for compiled C language programs, and is based on extensive measurements of the static and dynamic characteristics of "real" C language programs.

COMPILER OPTIONS

All options recognized by the **dmdcc** command are listed in Figure 6-1 and on the **dmdcc** manual page in the *5620 Dot-Mapped Display Reference Manual*. This section provides additional information for those options not completely described in the figure.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations, such as relocatable object files (**-c** option), assembly source expansions for C language code (**-S** option), or the output of the preprocessor (**-P** option). In general, the intermediate files may be saved and later resubmitted to the **dmdcc** command, with other files or libraries included as necessary.

When compiling C language source files, the most common practice is to use the **-c** option to save relocatable files. Subsequent changes to one file do not then require that the others be recompiled. A separate call to **dmdcc** without the **-c** option then creates the linked executable **dmda.out** file. A relocatable object file created under the **-c** option is named by adding a **.o** suffix to the source file name.

The **-W** option is used to specify options for each step that is normally invoked from the **dmdcc** command line. These steps are preprocessing, the first pass of the compiler, the second pass of the compiler, optimization, assembly, and link editing. At this time, only assembler and link editor options can be used with the **-W** option. The most common example of use of the **-W** option is "**-W***a,-m*", which passes the **-m** option to the assembler.

Option	Argument	Description
-c	none	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled. Object file name is *file.o*.
-g	none	Produce symbolic debugging information.
-o	*outfile*	Produce an output object file by the name *outfile*. The name of the default object file is **dmda.out**. NOTE: This is an option to the DMD link-editor, **m32ld**.
-D	*identifier*[=*constant*]	Define the external symbol *identifier* to the preprocessor, and give it the value *constant* (if specified). See Note at end of figure.
-E	none	Same as the **-P** option except output is directed to the standard output.
-I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in " " are searched for first in the directory of the file being compiled, then in directories named by the **-I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <> , the directory of the *file* argument is not searched. See Note at end of figure.

Figure 6-1. dmdcc Command Line Options (1 of 2)

Option	Argument	Description
-J	none	Compile this program for execution in the **stand-alone** environment, not **layers**.
-O	none	Invoke an object code optimizer.
-P	none	Suppress compilation and loading; that is, invoke only the preprocessor and leave the output on corresponding files suffixed **.i.**
-S	none	Compile the named C language programs and leave the assembly-language output on corresponding files suffixed **.s.**
-U	*identifier*	Undefine the named *identifier* to the preprocessor. See Note.
-V	none	Print the version of the assembler that is invoked.
-W	*c,arg1*[,*arg2*...]	Pass along the argument(s) *argi* to pass *c*, where *c* is one of [**p**, **0**, **1**, **2**, **a**, or **l**], indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively. See Note.

Note: The argument is appended to option with no embedded blanks.

Figure 6-1. dmdcc Command Line Options (2 of 2)

When the **-P** option is used, the compilation process stops after only preprocessing, with output left on *file.i*. This file will be unsuitable for subsequent processing by **dmdcc**.

If optimization is desired, select one of the two optimization options offered by **dmdcc**. The **-O** option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

The **-g** option produces information for the debugger, **dmdebug**.

REGISTER USE

Because a high-level language frees the programmer from involvement with the details of hardware architecture, most applications should not require access to registers. An exception is *register* variables. **Dmdcc** can assign up to six register variables; their use may result in smaller and faster programs.

For most uses, the implementation details of registers are not needed by programmers.

Registers can be accessed through an assembler escape, although this practice is not recommended. Registers have the following usage in the compiler:

REGISTER	USE
sp	Stack Pointer
fp	Frame Pointer
ap	Argument Pointer
r8-r3	Register variables
r2-r0	Scratch registers

The six register variables allowed by **dmdcc** are assigned to registers r8 through r3 in descending order. If more than six registers are declared in a C language source program, the compiler silently assigns data space instead.

Register r0 holds the return value of a function call. For a function returning a structure, register r2 passes the address to which the returned structure value should be stored. Function calls are assumed to require all scratch registers; therefore, any such registers that contain temporary variables at the time of a function call are automatically saved by the calling instruction in the local variable area on the stack before issuing the call.

C LANGUAGE

The C language used by the WE-32000 microprocessor has many features to accommodate both systems and general-purpose programming. The version of C language implemented closely resembles the C language described in the Kernighan and Ritchie book* with two major differences

* B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, (Englewood Cliffs, New Jersey: Prentice-Hall, 1978)

that are described in this manual. The differences are: the **dmdcc** compiler lacks the **float** and **double** data type specifiers, and **dmdcc** includes recent enhancements to the C language. These enhancements allow structures to be assigned or copied as a unit, and to be passed to and returned from functions.

On the WE-32000 microprocessor, C language data types map in the natural way for a 32-bit processor— that is, **char** maps to the type byte, **int** and **long** map to word, and **short** to halfword.

Standard C language leaves identification of the assembler escape keyword *asm* to the system designer. *Asm* has been implemented for **dmdcc** with the syntax *asm*(" assembly instruction"). For example, *asm*(" MOVW &0, r0") loads register zero with a zero. The assembly language instruction contained within the quotation marks is transmitted unchanged to the assembler.

> ***Note:*** Floating point is not supported by **dmdcc**. If the **float** or **double** data type specifiers are used, the named variable will be treated as an **int**, and a warning message will be issued. However, **float** and **double** remain reserved words.

Chapter 7

UTILITIES

PAGE

Chapter 7

UTILITIES

The output file from the **m32as** assembler and the **m32ld** link editor is an object file named **dmda.out**. It has a format called the Common Object File Format (COFF). The object file is executable if no errors or unresolved references are found. The file contains a header with size information, program sections, and a symbol table. Each section is composed of a section header, data, and relocation and line number information. Depending on the assembler or link editor options used to produce the object file, the file may be devoid of relocation entries, line number entries, the symbol table, or compiler-generated symbols.

The SGS provides a variety of utilities to read and manipulate object files. Among the functions performed by the utilities are listing, reducing, or deleting various parts of an object file or symbol table. A library consisting of 24 object modules that aid in the development of application programs is also provided as part of the SGS. The routines and data declarations that comprise the libraries will be used in common by many projects. This library approach allows efficient, controlled development of common code and enhances portability.

Most of this chapter consists of detailed descriptions of each utility. The utilities are:

m32conv Converts WE-32000 microprocessor object files from one host machine format to another host machine format.

m32cprs Compresses object files by removing duplicate structure and union descriptors.

m32dis Disassembles object files to allow assembly level debugging.

m32dump Dumps selected parts of the named object files.

m32list Produces a C language source list with line numbers that specify where breakpoints can be inserted.

m32lorder Generates an ordered listing of object files suitable for link editing in one pass, as done by **m32ld**.

m32nm Prints the symbol table for an object file.

m32size Reports the number of bytes of text, uninitialized data, and initialized data (and their sum) included in an object module.

m32strip Reduces file storage overhead by removing symbolic testing information from an object file.

Because the SGS runs under the UNIX Operating System, the utilities can use the many features of UNIX System Shell commands. For example, I/O redirection, pipes and filters, and the asynchronous capability of the shell are commonly used with the utilities. The defining of procedures and shell variables, the use of metacharacters, or other features of the shell may also prove useful.

ERROR MESSAGES

One error message common to many utilities is **can't open file**, meaning a file cannot be read. The message is usually caused by misspelling the file name, or the file is located in the wrong directory. A list of other commonly encountered error messages can be found under each utility description.

M32conv

Whenever a file is moved from one machine to another of different architecture, **m32conv** should be used to format the resulting file.

Differences in byte ordering and data formats cause object file formats to differ in their symbolic information when produced on machines of disparate architectures. **M32conv** converts a WE-32000 microprocessor object file (for example, **dmda.out**) from the internal format of one machine architecture to that of another architecture. For example, to move a WE-32000 microprocessor object file produced on an AT&T 3B20S minicomputer to a VAX*-11/780 minicomputer, one must use **m32conv** or the resulting file will not be in a usable format. **M32conv** does not alter the contents of the **.text** or the **.data** sections; it only modifies the headers, symbol tables, and other symbolic information.

File conversion is necessary and effective between machines of the following three architectures:

1. A DEC*-style byte ordering with 16-bit word length (for example, PDP*-11/70 Computer).

* Registered trademarks of Digital Equipment Corporation

2. A DEC-style byte ordering with 32-bit word length (for example, VAX-11/780 Computer).

3. An IBM-style byte ordering with 32-bit word length (for example, AT&T 3B20 Computer).

The output of **m32conv** is a file having the same name as the input file with a suffix of **.v**. Output cannot be redirected from the **m32conv** command.

M32conv is best used within a procedure for sending object files from one machine to another. Attempting to convert a file when no conversion is necessary results in an error message, although the input file is copied to the output file.

To use **m32conv**, enter the SGS command

m32conv [**-**] [**-p**] [**-s**] **-t target** *files*

where the **-t** option with a target name MUST be specified, and *files* is a list of files to be converted.

The minus sign (**-**) specifies that filenames are taken from the standard input. The **-s** option causes **m32conv** to function exactly as the UNIX system command **swab**, which exchanges adjacent odd- and even-numbered bytes. This may be useful depending on the actual transfer method used and the byte-ordering of the host machine. The **-p** option produces a file in portable archive format.

All diagnostics are intended to be self-explanatory. Fatal errors on the command line cause the program to terminate. Fatal errors within an input file cause the program to continue at the next input file.

M32cprs

The utility **m32cprs** reduces the size of a WE-32000 microprocessor object file by removing duplicate structure and union descriptors. To invoke this utility, enter the SGS command

m32cprs *options file1 file2*

where *file1* is the input file, *file2* is the output file, and the available *options* are **-p** and **-v**. The input file is not changed by this process; the output file, where the compressed version of the input is placed, must be specified by the user.

The **-p** option causes the printing of statistical messages including: total number of tags, total duplicate tags, and total reduction of the size of *file1* . The **-v** option causes verbose error messages to be printed if an error occurs.

Some of the most commonly encountered error messages are:

usage: m32cprs [-v] [-p] *infile outfile*

Caused by failure to specify names for both input and output files.

Infile has incorrect magic number

error condition: no compression

Occurs when *infile* is not in WE-32000 microprocessor object file format.

no duplicate tags

unable to open *infile*

unable to create *outfile*

M32dis

The WE-32000 microprocessor disassembler, **m32dis**, produces an assembly language listing of each input object file. The listing has a two-column format with assembly language statements in the right column and the corresponding binary object code and machine address of the code in the left column.

The disassembler produces a facsimile of the assembly language file that was assembled to produce a given object file. **M32dis** provides a convenient method to obtain a WE-32000 microprocessor assembly language listing of C language source programs and of assembly language programs written in Instruction Set 25 (IS25). Because the **dmdcc** compiler outputs IS25 code, programmers often need the disassembler to get listings in WE-32000 microprocessor assembly language.

To invoke the disassembler, enter the SGS command:

m32dis *options files*

where *options* are chosen from Figure 7-1, and *files* represents a list of object files. The default is for all sections containing text to be disassembled.

Option	Argument	Description
-d	section	Disassemble the named section as data, and print the offset of the data from the beginning of the section.
-da	section	Disassemble the named section as data, and print the actual address of the data.
-l	string	Disassemble the library file specified by *string*. For example, one would issue the command "**m32dis** -l x -l z" to disassemble the libraries **libx.a** and **libz.a**. The libraries are assumed to be in the SGS *LIB* directory. Consult your site administrator to check this.
-o	none	Print numbers in octal; default (without this option specified) is hexadecimal.
-t	section	Disassemble the named section as text.
-F	string	Disassemble the single function named *string* in each object file specified on the command line.
-L	none	Invoke a look-up of C source labels in the symbol table for subsequent printing.
-V	none	Print the version number of the disassembler being executed.

Note: Arguments are appended to options with no embedded blanks, except for the **-l** option.

Figure 7-1. M32dis Command Line Options

Three features of the **m32dis** listing are:

1. The disassembler prints line numbers for each C source line where a breakpoint can be set in square brackets (for example, "[5]" shows the fifth line where execution can be halted for debugging). The line numbers appear in the first column, one line above the instruction corresponding to the line where a breakpoint can be inserted.

2. The disassembler prints C function names followed by parentheses (for example, "printf( )" for the function **printf**). The function names appear in the first column, one line above the instruction that begins the function.

3. The disassembler prints computed addresses within a section when control is to be transferred to those addresses. They are printed within triangular brackets (for example, "<40>" is computed address 40). These addresses appear in the operand field of control transfer instructions following a relative displacement. The computed address is the sum of the relative displacement and the address of the instruction currently being disassembled.

Note that items 1 and 2 occur only if the information exists in the object file (for example, the code was compiled with the **-g** option given to **m32cc** and the information was not stripped out by a utility or link editor option).

Disassembler Options

The **-d** option causes the named section of the object file to be disassembled as a data section. The binary object code and its address relative to the beginning of the section are listed. **M32dis** makes no attempt to determine the corresponding assembly language statement. Addresses relative to the beginning of the named section are printed on the left side. The bytes of object code are printed on the right side, eight bytes per line.

The **-da** option causes disassembly of the named section of the object file as a data section. The binary object code and its absolute addresses are listed. No attempt is made to determine the corresponding assembly language statement.

The **-t** option causes the named section of the object file to be disassembled as a text section. The listing consists of the binary object code, its machine address, and the assembly language statements that produced the code. For example, if the SGS command is:

m32dis *-t section files*

then the bytes of object code are assumed to be opcode and operand encodings. The opcodes are looked up in the opcode disassembly table, and the operands are disassembled and printed.

The **-L** option causes the disassembler to print C source labels. This option also invokes an object file symbol table look-up to determine labels for subsequent printing. The C source label is printed on the line preceding the machine instruction that corresponds to the C statement with that label.

M32dis Error Messages

The following is a list of error messages commonly encountered while executing the disassembler:

m32dis: <*filename*>**: CANNOT OPEN:**

Means the input file cannot be read.

m32dis: <*filename*>**: BAD MAGIC** <*number*>

The input file is not a WE-32000 microprocessor object file.

m32dis: <*filename*>**: BAD ARCHIVE HEADER**

Occurs when the **-L** option is used and the disassembler cannot read the referenced library's archive header.

m32dis: *<filename>*: **CANT FIND SECTION** *<section name>*

An unknown section has been specified by the **-t** or **-d** option.

m32dis: *<filename>*: **CANT FIND SECTION HEADER** *<section name>*

The input file is not a WE-32000 microprocessor object file or the file was not properly converted using **m32conv**.

m32dis: BAD FLAG *<flag>*

An unrecognized option has been specified.

m32dis: PREMATURE EOF

m32dis: QUIRK--DATA SECTION HAS LINE NUMBER ENTRIES

If the disassembler cannot find an opcode in the disassembler opcode lookup table, the following message is printed on the same line as the bad object code:

ERROR UNKNOWN OPCODE

The disassembler will then attempt to resynchronize itself. There are three possible cases:

1. The file has been stripped of line number information as well as the symbol table. The following message will be printed:

   NO LINE NUMBER ENTRIES EXIST
   NO SYMBOL TABLE EXISTS
   FOLLOWING DISASSEMBLY MAY BE OUT OF SYNC

   The disassembler will then continue with the two bytes immediately following the bad opcode.

2. The file has been stripped of line number entries but has a symbol table. The following message will be printed:

```
NO LINE NUMBER INFORMATION EXISTS
DUMP TO NEXT FUNCTION OR SECTION END
IN ATTEMPT TO RESYNCHRONIZE
```

   The disassembler will then dump bytes of object code until the next function or the section end (whichever comes first) is reached. At that point the disassembler prints out the message:

```
DISASSEMBLER RESYNCHRONIZED
```

3. The file has line number entries. The disassembler then dumps bytes of object code until it reaches the next line where a breakpoint can be inserted. At that point the disassembler prints the message:

```
DISASSEMBLER RESYNCHRONIZED
```

M32dump

The **m32dump** utility allows examination of an object file by listing the contents of the file. The dump utility is normally used to look at different parts of an object file, with the parts being selected by options. **M32dump** attempts to format the information it dumps in a meaningful way by printing certain information in ASCII, hexadecimal, octal, or decimal as appropriate. The input file is unchanged after execution of **m32dump**, and no new files are created. **M32dump** accepts as input both object files and archive libraries of object files. Figure 7-2 lists **m32dump** options.

Option	Argument	Description
-a	none	Dump the archive header of each member of each input archive file.
-d	number	Dump the section number given or dump the range of sections beginning with the given number and ending either at the last section or at the number specified by +**d**.
+d	number	Dump only those sections having section numbers less than *number*. Begin either with the first section or with the section specified by the **-d** option.
-f	none	Dump each file header.
-h	none	Dump all section headers.
-l	none	Dump line number information.
-n	name	Dump only the information pertaining to the named entity. This option is used with **-h, -s, -r, -l,** and **-t**.
-o	none	Dump each optional header.
-r	none	Dump relocation information.
-s	none	Dump section contents.
-t	none	Dump symbol table entries.
-t	index	Dump only the indexed symbol table entry. The **-t** used with the +**t** option specifies a range of symbol table entries.

Figure 7-2. M32dump Command Line Options (1 of 2)

Option	Argument	Description
+t	index	Dump symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **-t** option.
-v	none	Print symbolic, rather than numeric, information.
-z	name	Dump line number entries for the named function.
-z	name,number	Dump line number entry or range of line numbers starting at number for the named function.
+z	number	Dump line numbers starting at either the function name or number specified by **-z** up to number specified by **+z**.

Figure 7-2. M32dump Command Line Options (2 of 2)

The options **-a, -f, -h, -l, -o, -r, -s, -t,** and **-z** used with a named argument specify which parts of an object file are to be dumped. These are the basic options and can be used independently; others are modifying options. The options **-d, +d, -n, -t, +t** used with an argument, and **-z, +z** with a numerical argument are used in combination with other options to limit the range and type of information that is to be printed. The **-v** option is used to modify all but the **-o** and **-s** options. The **-v** option causes **m32dump** to interpret the information and print symbols instead of numbers; for example, "static" instead of "0x03".

Blanks separating an option and its modifier are optional. The comma separating the name from the number modifying the **-z** option may be replaced with a blank.

A simple example of an **m32dump** is the SGS command

m32dump -t *dmda.out*

which would display the symbol table from the file *dmda.out*. The command

m32dump -tv *dmda.out*

displays the symbol table from the file *dmda.out* in symbolic form. The command

m32dump -f -h -r -t 3 +t 10 *test.o* **>testdump**

lists the file and section headers, the relocation information, and the symbol table entries three through ten for the object file "test.o". This command also places the output on the file "testdump".

The more common error messages produced by **m32dump** are:

usage: m32dump [flags] file ...

Caused by neglecting to name the object file that is to be dumped.

m32dump: bad magic *file.out*

Occurs when the file *file.out* is not a WE-32000 microprocessor object file.

m32dump: cannot open *file.out*

Means *file.out* cannot be read.

m32dump: unknown option *OPTION*

M32list

The utility **m32list** lists C language source files with line number information attached. **M32list** uses the object file corresponding to the input C language source file to determine the lines where breakpoints can be set. Generally breakpoints can be set at each executable statement that begins a new line of source code.

To invoke the WE-32000 microprocessor list utility, use the command

m32list [V] [-h] *source* [**source**...] [**object**]

where the square brackets denote optional entries, *source* is the source file name, and *object* is the object file name. If several C source files were used to create the object file, then a list of source files should be input to **m32list**. The last name in the list of files is considered to be the name of the object file. The default object file, **dmda.out**, is used when no object file appears on the command line. The input object file MUST have symbolic debugging symbols for **m32list** to work.

Line numbers are printed for each compiler-generated line where a breakpoint can be inserted. Line numbering begins anew for each C language function. Line number 1 always indicates the line containing the left curly brace ({) that begins a function body. Line numbers are also printed for inner block redeclarations of local variables so that those variables can be distinguished by the symbolic debugger.

The **-h** option suppresses the printing of headings.

Object files that have no line numbers cause an error message to be printed. Because **m32list** does not use the C preprocessor, it may not recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

Some errors commonly encountered when using **m32list** are:

usage: m32list sourcefile [sourcefile...] [object file]

Caused when no object file or no source file is specified.

m32list: name: cannot open

Caused when an input file name cannot be read.

m32list: unknown option *option*

An illegal option was typed.

M32lorder

The link editor, **m32ld**, like most loaders, requires that object file libraries provided to it are in a specified order. The order must be such that every symbol and function is referenced before it is defined. The library order utility, **m32lorder**, can determine this required order.

The SGS command for library ordering works the same way as its UNIX system counterpart. To invoke the library ordering utility, use the SGS command:

m32lorder *files*

where *files* indicates the input of one or more object or library archive files. The output of **m32lorder** is a list of pairs of object file names, where the first file of the pair contains references to external identifiers defined in the second. Therefore, the second member of the pair must appear after the first to be properly loaded.

The names of input object files *must* end with **.o**, even when contained in library archives. Files with names not adhering to this rule have their global symbols and references attributed to some other file, and nonsense results.

The output from **m32lorder** may be processed by the UNIX system command **tsort** to find an ordering of a library suitable for one-pass access by the link editor, **m32ld**. The following example shows the use of **tsort**, along with the UNIX system archive maintainer, **ar**, to build a new library from all existing files with names ending in **.o**. The archive library is named *libx.a* both before and after the operation.

```
ar cr libx.a `m32lorder *.o | tsort`
```

M32nm

The name list utility, **m32nm**, displays the symbol table for each WE-32000 microprocessor object file that is given as input. The input may be a relocatable or an absolute WE-32000 microprocessor object file; or it may be an archive library of relocatable or absolute object files.

For each symbol in the table, the following information is printed:

Name — the name of the symbol.

Value — the symbol value expressed as an offset or an address depending on storage class.

Class — its storage class.

Type — its type and derived type. If the symbol is an instance of a structure or of a union, then the structure or union tag is given following the type (for example, "struct-person" where "person" is the structure tag). If the symbol is an array, then the array dimensions are given following the type (for example, char[n][m]).

Size — its size in bytes, if applicable. Special symbols have undefined size.

Line — the source line number where it is defined, if applicable.

Section for storage classes static and external, the object file section containing the symbol.

m32nm does not change the input file and produces no new file. The syntax to invoke the name list utility is:

m32nm *options filenames*

where *options* are chosen from Figure 7-3 and *filenames* are the names of the input file(s) and/or archive(s).

Option	Description
-a	Produces full output. Redundant symbols are not suppressed.
-d	Symbol value and size is printed in decimal instead of hexadecimal.
-e	Print only static and external symbols.
-f	Produce "fancy" output by post-processing the symbol table information to reflect the block structure of the source code.
-n	Sort the external symbols by name before printing them.
-o	Print the value and size for each symbol in octal instead of the normal hexadecimal.
-u	Print only the undefined symbols.
-v	Sort external symbols by value before printing them.
-V	Print the version name of **m32nm** that is executing.

Figure 7-3. M32nm Command Line Options

The options may be specified in any order, either singly or in combination, and may appear anywhere on the command line. Therefore, both "**m32nm** *name -e -v*" and "**m32nm** *-ve name*" print the static and external symbols in *name*, with the external symbols sorted by value. If neither the **-n** nor the **-v** option is specified, the external symbols are printed in the order in which they are encountered.

Some common error messages that **m32nm** produces are:

usage: m32nm: *file* **: bad magic**

Input file is not a WE-32000 microprocessor object file.

m32nm: name: cannot open

Input file cannot be read.

m32nm: name: bad magic

Input file is not a WE-32000 microprocessor object file.

m32nm: name: no symbols

Symbols were stripped from the input file before it was input to **m32nm**.

m32nm: unknown option *option*

M32size

M32size prints the number of bytes required for each section (**.text**, **.data**, and **.bss**) of the input WE-32000 microprocessor object file and the total number of bytes for all three sections. Such information may be needed for locating sections in memory.

The file input to **m32size** remains unchanged. The output consists of the name of each section, followed by its size in bytes, its physical address, and its virtual address. The form of the SGS command for **m32size** is:

m32size [**-o**] [**-d**] [**-V**] *filename[s]*

By default, numbers are printed in hexadecimal. The **-d** option specifies decimal numbers; the **-o** option specifies octal. Version information is printed when the **-V** option is specified.

Commonly encountered diagnostics are:

m32size: *filename* **: cannot open**

When *filename* cannot be read.

m32size: *filename* **: bad magic**

When *filename* is not a WE-32000 microprocessor object file.

M32strip

The strip utility removes the symbol table and line number information from WE-32000 microprocessor object files and archive libraries, thus saving space. The effect of **m32strip** is the same as the **-s** option of **m32ld**. After a file has been stripped, no symbolic debugging access is available for that file. This command should be run only on production versions of object files that have been debugged and tested.

The SGS command to strip is:

m32strip [**-l**] [**-x**] [**-r**] [**-V**] *name...*

where *name* is the name of a WE-32000 microprocessor object file or archive. Any number of *names* may be specified. If *name* is an archive, **m32strip** removes the local symbols from each object module in the archive. By deleting these symbols, the size of the archive is decreased and link editing performance improves.

The amount of information stripped from the symbol table can be controlled by using either the **-l** or the **-x** options. With the **-l** option, only line number information is stripped. Symbol table information remains unchanged. With the **-x** option, no static or external symbol information gets stripped. The **-V** option prints version information.

If there are any relocation entries in the object file and symbol table information is to be stripped, **m32strip** terminates without stripping *name* and prints the error message:

m32strip: *name* **: relocation entries present; cannot strip**

The **-r** option allows the user to override this warning and force **m32strip** to strip an object module even if the module contains relocation information. Because the relocation information is needed by the link editor, use of **-r** option is not recommended. Certain systems, however, may find it useful.

Other commonly encountered error messages are:

m32strip: *name* **: cannot open**

when *name* cannot be read.

m32strip: *name*: **bad magic**

when *name* is not a WE-32000 microprocessor object file.

usage: m32strip [-l] [-x] [-r] *file...*

when no input file has been specified.

APPENDIX A

VECTOR TABLES

This appendix contains a listing of the ROM-resident library routines in the DMD with entry points. The *Stand-Alone Vector Table* contains library routines associated with the **stand-alone** environment. The *Layers Vector Table* contains library routines associated with the **layers** environment.

STAND-ALONE VECTOR TABLE

NAME	INDEX	NAME	INDEX
0	Firm[100]	balloc	Firm[11]
0	Firm[162]	baud_speeds	Firm[153]
0	Firm[163]	bfree	Firm[12]
0	Firm[170]	binit	Firm[13]
0	Firm[171]	bitblt	Firm[14]
0	Firm[172]	blocked	Firm[125]
0	Firm[182]	botbits	Firm[152]
abs	Firm[0]	bramgetstr	Firm[15]
aciagetc	Firm[197]	bramputstr	Firm[16]
aciainit	Firm[2]	cbufs	Firm[126]
aciapaws	Firm[1]	ceil	Firm[17]
aciapoke	Firm[196]	checkbram	Firm[18]
aciaputc	Firm[198]	cur	Firm[127]
aciarxint	Firm[217]	cursallow	Firm[19]
aciatrint	Firm[3]	cursblt	Firm[20]
acioctl	Firm[199]	cursinhibit	Firm[21]
add	Firm[4]	cursinit	Firm[22]
addr	Firm[5]	cursor	Firm[128]
alloc	Firm[6]	cursset	Firm[23]
allocb	Firm[122]	cursswitch	Firm[24]
allocendp	Firm[123]	curtabp	Firm[129]
allocinit	Firm[7]	C_arrows	Firm[161]
allocstartp	Firm[124]	C_confirm	Firm[167]
auto1	Firm[8]	C_crosshair	Firm[159]
auto2	Firm[9]	defont	Firm[130]
auto4	Firm[10]	dellayer	Firm[25]
aux1inch	Firm[192]	devtab	Firm[210]
aux1ops	Firm[213]	disconnect	Firm[26]
aux1outch	Firm[193]	display	Firm[131]
aux2inch	Firm[194]	div	Firm[27]
aux2ops	Firm[214]	downback	Firm[185]

STAND-ALONE VECTOR TABLE

NAME	INDEX	NAME	INDEX
aux2outch	Firm[195]	dtr	Firm[132]
ENDAREA	Firm[133]	kbdchar	Firm[54]
eqpt	Firm[28]	kbdinit	Firm[55]
eqrect	Firm[29]	kbdrepeat	Firm[136]
excep_int	Firm[30]	kbdrpt	Firm[157]
excep_msg	Firm[34]	kbdstatus	Firm[137]
excep_norm	Firm[31]	kgetc	Firm[56]
excep_proc	Firm[32]	layerop	Firm[57]
excep_stack	Firm[33]	lback	Firm[138]
floor	Firm[35]	lbitblt	Firm[58]
free	Firm[36]	lblt	Firm[59]
freeall	Firm[37]	Lbox	Firm[60]
freelist	Firm[134]	lfront	Firm[139]
gcalloc	Firm[38]	Lgrey	Firm[61]
gcfree	Firm[39]	load	Firm[62]
gcfreeall	Firm[40]	logports	Firm[187]
gcinit	Firm[41]	lpoint	Firm[63]
getnum	Firm[42]	lrectf	Firm[64]
hostgetc	Firm[189]	lsegment	Firm[65]
hostops	Firm[211]	ltexture	Firm[66]
hostputc	Firm[188]	main	Firm[67]
inset	Firm[43]	maxaddr	Firm[140]
interrupt	Firm[135]	MCur_clock	Firm[160]
jline	Firm[44]	MCur_cup	Firm[168]
jlineto	Firm[45]	MCur_deadmouse	Firm[169]
jmove	Firm[46]	MCur_skull	Firm[164]
jmoveto	Firm[47]	MCur_sweep	Firm[165]
jpoint	Firm[48]	MCur_target1	Firm[166]
jrectf	Firm[49]	menuhit	Firm[68]
jsegment	Firm[50]	mouse	Firm[141]
jstring	Firm[51]	msvd_hndl	Firm[219]

STAND-ALONE VECTOR TABLE

NAME	INDEX	NAME	INDEX
jstrwidth	Firm[52]	mul	Firm[69]
jtexture	Firm[53]	nap	Firm[70]
NAVAIL	Firm[142]	Rect	Firm[88]
newlayer	Firm[71]	rectclip	Firm[89]
nextlong	Firm[143]	rectf	Firm[90]
NLONGS	Firm[144]	rectXrect	Firm[91]
obsclean	Firm[218]	remote	Firm[147]
own	Firm[72]	ringbell	Firm[92]
pfedit	Firm[73]	Rpt	Firm[93]
piohint	Firm[184]	rsubp	Firm[94]
point	Firm[74]	savecur	Firm[148]
polyf	Firm[181]	sccAgetc	Firm[201]
prcvchar	Firm[186]	sccAioctl	Firm[203]
prntops	Firm[212]	sccApoke	Firm[200]
probe_io	Firm[216]	sccAputc	Firm[202]
psendchar	Firm[156]	sccBgetc	Firm[205]
Pt	Firm[75]	sccBioctl	Firm[207]
PtCurrent	Firm[145]	sccBpoke	Firm[204]
ptinpoly	Firm[183]	sccBputc	Firm[206]
ptinrect	Firm[76]	sccinit	Firm[215]
qclear	Firm[77]	screenswap	Firm[95]
qgetc	Firm[78]	scroll	Firm[96]
qinit	Firm[79]	segment	Firm[97]
qputc	Firm[80]	sendbreak	Firm[98]
qputstr	Firm[81]	sendchar	Firm[99]
qsetbram	Firm[220]	setbram	Firm[101]
qsort	Firm[180]	setup	Firm[102]
queues	Firm[146]	SetupFlag	Firm[158]
raddp	Firm[82]	sopbaud	Firm[209]
rcvchar	Firm[83]	sopoke	Firm[208]
realalloc	Firm[84]	spl0	Firm[103]

STAND-ALONE VECTOR TABLE

NAME	INDEX	NAME	INDEX
realballoc	Firm[85]	spl1	Firm[104]
realgcalloc	Firm[86]	spl4	Firm[105]
reboot	Firm[87]	spl5	Firm[106]
spl6	Firm[107]	tstdcd	Firm[118]
spl7	Firm[108]	T_background	Firm[173]
splx	Firm[109]	T_black	Firm[178]
stdcurtab	Firm[149]	T_checks	Firm[179]
stopbreak	Firm[190]	T_darkgrey	Firm[174]
stopdisc	Firm[191]	T_grey	Firm[175]
string	Firm[110]	T_lightgrey	Firm[176]
strlen	Firm[111]	T_white	Firm[177]
strwidth	Firm[112]	ublocked	Firm[150]
sub	Firm[113]	upfront	Firm[119]
test32	Firm[114]	usercurtab	Firm[151]
texture	Firm[115]	version	Firm[120]
topbits	Firm[154]	ver_str	Firm[155]
trdisable	Firm[116]	wait	Firm[121]
trenable	Firm[117]		

LAYERS VECTOR TABLE

NAME	INDEX	NAME	INDEX
0	Sys[196]	control	Sys[97]
0	Sys[197]	copy	Sys[98]
0	Sys[204]	crc	Sys[99]
0	Sys[205]	cup	Sys[167]
0	Sys[206]	Current	Sys[77]
0	Sys[215]	cursallow	Sys[68]
abs	Sys[223]	curse	Sys[100]
add	Sys[2]	cursinhibit	Sys[67]
addr	Sys[3]	cursswitch	Sys[237]
agent	Sys[89]	C_confirm	Sys[201]
alarm	Sys[59]	C_crosshair	Sys[193]
allocendp	Sys[249]	debug	Sys[55]
allocstartp	Sys[250]	debugger	Sys[169]
arrows	Sys[195]	defont	Sys[1]
bfree	Sys[6]	Delete	Sys[78]
bldargs	Sys[90]	dellayer	Sys[11]
boot	Sys[91]	delproc	Sys[101]
botbits	Sys[232]	demux	Sys[102]
Bottom	Sys[75]	display	Sys[234]
boxcurs	Sys[184]	div	Sys[12]
bramgetstr	Sys[224]	dobutton	Sys[103]
bttns	Sys[92]	doctl	Sys[104]
bullseye	Sys[182]	downback	Sys[220]
canon	Sys[93]	ENDAREA	Sys[251]
cbufavail	Sys[221]	eqpt	Sys[226]
ceil	Sys[225]	eqrect	Sys[13]
checkbram	Sys[236]	error	Sys[105]
checkrect	Sys[94]	excep_norm	Sys[106]
clear	Sys[54]	excep_proc	Sys[107]
clippt	Sys[95]	excep_stack	Sys[108]
clockroutine	Sys[96]	exec	Sys[109]

LAYERS VECTOR TABLE

NAME	INDEX	NAME	INDEX
Control	Sys[76]	execsw	Sys[110]
Exit	Sys[79]	max	Sys[123]
floor	Sys[227]	maxaddr	Sys[252]
free	Sys[15]	MCur_clock	Sys[194]
freemem	Sys[111]	MCur_cup	Sys[202]
Free_RAM	Sys[163]	MCur_deadmouse	Sys[203]
gcfree	Sys[57]	MCur_skull	Sys[198]
getanum	Sys[112]	MCur_sweep	Sys[199]
getchar	Sys[113]	MCur_target1	Sys[200]
getlong	Sys[114]	menufn	Sys[176]
getrect	Sys[58]	menuhit	Sys[34]
givemouse	Sys[115]	menutext	Sys[185]
host_int	Sys[116]	min	Sys[124]
hst_init	Sys[181]	mouse	Sys[253]
inset	Sys[16]	move	Sys[125]
itox	Sys[117]	Move	Sys[80]
jstring	Sys[118]	mpxcore	Sys[126]
jstrwidth	Sys[74]	mpxdelwind	Sys[127]
kbdlist	Sys[179]	mpxkbdchar	Sys[128]
kbdlistp	Sys[180]	mpxkbdflush	Sys[129]
kbdproc	Sys[168]	mpxnewwind	Sys[71]
key_int	Sys[119]	mpxsendchar	Sys[130]
last	Sys[189]	mpxublk	Sys[131]
layerop	Sys[238]	msvid_int	Sys[132]
lbitblt	Sys[239]	mul	Sys[35]
lblt	Sys[240]	nap	Sys[28]
lpoint	Sys[60]	New	Sys[81]
lrectf	Sys[61]	newlayer	Sys[36]
lscroll	Sys[120]	newline	Sys[133]
lscrolx	Sys[121]	newproc	Sys[70]
lsegment	Sys[62]	newwindow	Sys[72]

LAYERS VECTOR TABLE

NAME	INDEX	NAME	INDEX
ltexture	Sys[63]	nextlong	Sys[254]
main	Sys[122]	nlcount	Sys[134]
numbers	Sys[135]	rectclip	Sys[42]
outline	Sys[137]	rectXrect	Sys[41]
out_int	Sys[136]	recvchars	Sys[145]
P	Sys[0]	Reply	Sys[83]
passchar	Sys[222]	reshape	Sys[146]
patch	Sys[191]	Reshape	Sys[84]
patchedspot	Sys[190]	restart	Sys[147]
pconfig	Sys[175]	Retry	Sys[85]
pconvs	Sys[174]	ringbell	Sys[165]
pinit	Sys[138]	Rpt	Sys[231]
point2layer	Sys[218]	rsubp	Sys[44]
precv	Sys[139]	rtransform	Sys[65]
proctab	Sys[173]	second	Sys[170]
psend	Sys[140]	sendbusy	Sys[186]
Psend	Sys[82]	sendnchars	Sys[47]
psendchar	Sys[192]	sendpkt	Sys[148]
pt	Sys[141]	setbram	Sys[244]
Pt	Sys[229]	setdata	Sys[149]
ptimeout	Sys[142]	shade	Sys[150]
ptinpoly	Sys[216]	shademap	Sys[187]
ptinrect	Sys[38]	shutdown	Sys[151]
qclear	Sys[241]	skull	Sys[183]
qgetc	Sys[242]	sleep	Sys[32]
qputc	Sys[243]	spl0	Sys[245]
qsort	Sys[235]	spl4	Sys[246]
raddp	Sys[39]	spl7	Sys[247]
readchar	Sys[143]	splx	Sys[248]
realgcalloc	Sys[56]	string	Sys[48]
realtime	Sys[66]	strlen	Sys[228]

LAYERS VECTOR TABLE

NAME	INDEX	NAME	INDEX
rebootflag	Sys[188]	strwidth	Sys[49]
RECT	Sys[144]	sub	Sys[50]
Rect	Sys[230]	sw	Sys[51]
Sw	Sys[86]	Ujpoint	Sys[22]
swinit	Sys[162]	Ujrectf	Sys[23]
switcher	Sys[152]	Ujsegment	Sys[24]
sysrun	Sys[166]	Ujstring	Sys[25]
ticks	Sys[171]	Ujtexture	Sys[26]
ticks0	Sys[172]	Ukbdchar	Sys[27]
tolayer	Sys[73]	Uown	Sys[37]
Top	Sys[87]	upfront	Sys[52]
topbits	Sys[233]	Upoint	Sys[29]
transform	Sys[64]	Upolyf	Sys[214]
trap	Sys[153]	Urcvchar	Sys[40]
T_background	Sys[207]	Urectf	Sys[30]
T_black	Sys[212]	Urequest	Sys[43]
T_checks	Sys[213]	Uscreenswap	Sys[45]
T_darkgrey	Sys[208]	Usegment	Sys[31]
T_grey	Sys[209]	Usendchar	Sys[46]
T_lightgrey	Sys[210]	usermouse	Sys[178]
T_white	Sys[211]	Utexture	Sys[33]
Ualloc	Sys[4]	Uwait	Sys[53]
Uballoc	Sys[5]	version	Sys[164]
Ubitblt	Sys[7]	whichaddr	Sys[219]
Uclipbttn	Sys[217]	whichbutton	Sys[154]
Ucursallow	Sys[8]	whichlayer	Sys[155]
Ucursinhibit	Sys[9]	whichproc	Sys[156]
Ucursset	Sys[69]	windowmenu	Sys[177]
Ucursswitch	Sys[10]	windowproc	Sys[157]
Uexit	Sys[14]	windowstart	Sys[158]
Ujinit	Sys[17]	writec	Sys[159]

LAYERS VECTOR TABLE			
NAME	**INDEX**	**NAME**	**INDEX**
Ujline	Sys[18]	zombexec	Sys[160]
Ujlineto	Sys[19]	zombsw	Sys[161]
Ujmove	Sys[20]	_start	Sys[88]
Ujmoveto	Sys[21]		

APPENDIX B

SAMPLE PROGRAMS

This appendix contains the C language source code for some of the demonstration (**demo**) programs. These programs are part of the DMD Core Software Package provided with the TELETYPE 5620 Dot-Mapped Display (DMD) terminal.

The source code presented in this appendix shows how programs are constructed to run on the DMD. The source code for *all* demo programs is located in:

$DMD/src/demo

The makefiles compile the demos for the **layers** and **stand-alone** environments. They are:

makefile

This is the master makefile. It will compile and install the demos for the **layers** and **stand-alone** environments. The makefiles **sa/makefile** and **layers/makefile** are called by this makefile.

sa/makefile

This is the makefile for the **stand-alone** environment. It will compile and install the **stand-alone** demos. If you are installing the demos for both environments (**layers** and **stand-alone**), use **makefile**.

layers/makefile

This is the makefile for the **layers** environment. It will compile and install the **layers** demos. If you are installing the demos for both environments (**layers** and **stand-alone**), use **makefile**.

The demos **a**, **b**, **bltdemo**, and **doodle** are easy to understand. Start with these demos first.

Source Code for Demo 'a'

```
/*  Copyright (c) 1984 AT&T  */
/*    All Rights Reserved    */

#include <sccsid.h>
VERSION(@(#)a.c  1.1);

#include <dmd.h>

#define  avg(a,b) (((a)+(b))>>1)
main()
{
  jinit ();
  request(KBD);
  cursinhibit ();
  ripple();
  cursallow ();
  exit();
}

Rectangle
canon(p1, p2)
Point p1, p2;
{
  register Rectangle r;

  r.origin.x = min(p1.x, p2.x);
  r.origin.y = min(p1.y, p2.y);
  r.corner.x = max(p1.x, p2.x);
  r.corner.y = max(p1.y, p2.y);
  return r;
}
ripple()
{
  register Point p1, p2;
  register Point inc;
```

```
  register Point p;

  p1.x=p2.x=avg(Drect.origin.x, Drect.corner.x);
  p1.y=p2.y=avg(Drect.origin.y, Drect.corner.y);
  inc.x=1;
  inc.y=1;
  rectf(&display, Drect, F_XOR);
  p = Drect.origin;
  while (kbdchar() != 'q') {
    rectf(&display, canon(p1, p2), F_XOR);
    if (p1.x==Drect.origin.x || p1.x==Drect.corner.x) {
      inc.x= -inc.x;
    }
    if (p1.y==Drect.origin.y || p1.y==Drect.corner.y) {
      inc.y= -inc.y;
    }
    p1 = sub (p1, inc);
    p2 = add (p2, inc);
    wait (CPU);
  }
}
```

Source Code for Demo 'ball'

```
/*  Copyright (c) 1984 AT&T  */
/*    All Rights Reserved    */

#include <sccsid.h>
VERSION(@(#)ball.c  1.1);

#include <dmd.h>

Point spot, v, a;
int Topx, Topy;
int Botx, Boty;
Bitmap *ball;
int  R;
long  R2;
main()
{
  request(KBD);
  R=32;
  R2=R*(long)R;
  ball=balloc(Rect(0, 0, 2*R+1, 2*R+1));
  sphere();
  P->state |= RESHAPED;
  while(kbdchar()!='q'){
    if (P->state & RESHAPED) {
      P->state &= ~RESHAPED;
      rectf(&display, Drect, F_CLR);
      Topx=10*Drect.corner.x-320;
      Topy=10*Drect.corner.y-320;
      Botx=10*Drect.origin.x+320;
      Boty=10*Drect.origin.y+320;
      a.x = 0;
      a.y = 1;
      spot.x = (Botx+Topx)/2;
      spot.y = Boty;
      v.x=5;
```

```
        v.y=0;
        drawball();
      }
      drawball();
      v = add(v, a);
      spot = add(spot, v);
      if(spot.x >= Topx) {
        spot.x = 2*Topx - spot.x;
        v.x = -v.x;
      }
      else if(spot.x <= Botx) {
        spot.x = 2*Botx - spot.x;
        v.x = -v.x;
      }
      if(spot.y >= Topy) {
        spot.y = 2*Topy - spot.y;
        v.y = -v.y;
      }
      else if(spot.y <= Boty) {
        spot.y = 2*Boty - spot.y;
        v.y = -v.y;
      }
      drawball();
      sleep(4);
    }
    bfree(ball);
    exit();
}
drawball()
{  Point p;
    p = sub(div(spot, 10), Pt(R, R));
    bitblt(ball, ball->rect, &display, p, F_XOR);
}
int  illum[3]={5, 4, 3};  /* illum=7 (pretty close) */
int  view[3]={1, 0, 0};
#define  DITHSIZE  8
#define  DITHMASK  (DITHSIZE-1)
```

```
int dith[DITHSIZE][DITHSIZE]={
   0,  32,   8,  40,   2,  34,  10,  42,
  48,  16,  56,  24,  50,  18,  58,  26,
  12,  44,   4,  36,  14,  46,   6,  38,
  60,  28,  52,  20,  62,  30,  54,  22,
   3,  35,  11,  43,   1,  33,   9,  41,
  51,  19,  59,  27,  49,  17,  57,  25,
  15,  47,   7,  39,  13,  45,   5,  37,
  63,  31,  55,  23,  61,  29,  53,  21,
};
sphere(){
  register x, y, z;
  register d;
  register I;
  rectf(ball, ball->rect, F_CLR);
  for(y= -R; y<=R; y++)  /* y across, pos to right */
    for(z= -R; z<=R; z++){  /* z pos up */
      if(z*z+y*y>R2)
        continue;
      x=sqrtryz(R, y, z);
      /* I=(illum.r)(view.r) */
      I=muldiv(x,  /* view.r */
         (x*5+y*4+z*3), /* illum.r */
          87);  /* a scale factor: ~RLV */
      if(I<=0)  /* unilluminated crescent */
        goto black;
      if(I>dith[y&DITHMASK][z&DITHMASK])
        point(ball, Pt(y+R, -z+R), F_CLR);
      else
    black:
        point(ball, Pt(y+R, -z+R), F_OR);
    }
}
```

Source Code for Demo 'bltdemo'

```
/*  Copyright (c) 1984 AT&T  */
/*    All Rights Reserved    */

#include <sccsid.h>
VERSION(@(#)bltdemo.c  1.1);

#include <dmd.h>
Rectangle myrect={400, 400, 501, 501};
main()
{
  register x, y;
  register long i;

  jinit();
  WonB();
  cursinhibit ();
  for(y=0; y<YMAX; y+=20){
    jmoveto(Pt(0, y));
    jlineto(Pt(XMAX, y), F_STORE);
  }
  for(x=0; x<XMAX; x+=20){
    jmoveto(Pt(x, 0));
    jlineto(Pt(x, YMAX), F_STORE);
  }
  for(x=0; x<300; x++){
    bitblt(&display, myrect, &display,
      sub(myrect.origin, Pt(1, 1)), F_STORE);
    myrect=rsubp(myrect, Pt(1, 1));
  }
```

```
  eor();
  for(i=200000; --i;) ;
  cursallow ();
  exit();
}
eor()
{
  register long i, *p;

  p=(long *)display.base;
  for(i=0; i<25*1024L; i++)
    *p++^=0xFFFFFFFF;
}
```

Source Code for Demo 'bounce'

```
/*  Copyright (c) 1984 AT&T  */
/*    All Rights Reserved    */

#include <sccsid.h>
VERSION(@(#)bounce.c  1.1);

#include <dmd.h>

#define N 25
Point foo[] = {
  312, 799,
  650, 650,
    7, -11
   -5,  3,
   13,  7,
    3, 11,
};
char *speedText[] = {
  "fast",
  "  .  ",
  "  .  ",
  "  .  ",
  "slow",
  NULL
};
Menu speedMenu = {
  speedText
};

main()
{
  register speed=0, item;
  register i = 0, j;
  Point from[N], to[N];
  Point dfrom, dto;
```

```
request(KBD|MOUSE);
P->state |= RESHAPED;
for (; kbdchar() != 'q'; sleep (speed)) {
  if (bttn1 ()) {
    if ((item = menuhit (&speedMenu, 1)) != -1) {
      speed = 1<<item;
    }
  }
  if (bttn3 ()) {
    request (KBD);  /* Unrequesting "MOUSE" */
    sleep (1);     /* allows the DMD to display */
    request (KBD|MOUSE);  /* the button 3 menu. */
  }
  if (P->state & RESHAPED) {
    if (!(P->state & MOVED)) {
      rectf (&display, Drect, F_XOR);
      for(j=0; j<N; j++){
        from[j].x=0; from[j].y=0;
        to[j].x=0; to[j].y=0;
      }
      from[0] = foo[0];
      from[1] = foo[0];
      to[0] = foo[1];
      to[1] = foo[1];
      dfrom = foo[2];
      dto = foo[3];
    }
    P->state &= ~(MOVED|RESHAPED);
  }
  j = i;
  if (++i >= N)
    i = 0;
  jsegment(from[i], to[i], F_XOR);
  from[i] = from[j];
  bump(&from[i], &dfrom);
  to[i] = to[j];
```

```
    bump(&to[i], &dto);
    jsegment(from[i], to[i], F_XOR);
  }
}

bump(p,dp)
register Point *p, *dp;
{
  if ((p->x += dp->x) > XMAX || p->x < 0) {
    dp->x = -dp->x;
    p->x += dp->x << 1;
  }
  if ((p->y += dp->y) > YMAX || p->y < 0) {
    dp->y = -dp->y;
    p->y += dp->y << 1;
  }
}
```

Source Code for Demo 'clock'

```
/*  Copyright (c) 1984 AT&T  */
/*    All Rights Reserved    */

#include <sccsid.h>
VERSION(@(#)clock.c  1.2);

#include <dmd.h>
#include <font.h>

#define  atoi2(p)  ((*(p)-'0')*10 + *((p)+1)-'0')
#define  itoa2(n, s)  { (*(s) = (n)/10 + '0');
              (*((s)+1) = n   10 + '0'); }
Point  ctr, p, cur;
int  dx, dy;
int  h, m, s;  /* hour, min, sec */
int  rh, rm, rs, rspread;
              /* radius of h m and s and spread ... */
int  ah = 0, am = 0, as = 0;  /* angle */
int  lah, lam, las;  /* last angles */
int  rad;
int  olds;
int  first = 1;

main(argc, argv)
char *argv[];
{
  char date[40], *p;
  register long oldtime;
  int stat, c, ds;

  request(KBD);


  rectf(&display, Drect, F_XOR);
  if(argc!=2){
```

```
    jmoveto(Pt(0, 0));
    jstring("Usage: 32ld clock
    sleep(600);
    exit();
  }
  initface();
  strcpy(date, argv[1]);
  h = atoi2(date+11);
  m = atoi2(date+14);
  s = atoi2(date+17);

  oldtime=realtime();
  for ( ds = 0;; ) {
    while (realtime() <= oldtime)
      sleep(20);
    ds += realtime()-oldtime;
    oldtime=realtime();
    s += ds / 60;
    ds  = 60;
    if (olds == s)
      continue;
    while (s >= 60) {
      s -= 60;
      m++;
    }
    olds = s;
```

APPENDIX C

RESERVED WORDS

This appendix lists the reserved words used by DMD software from AT&T. The reserved words are listed in alphanumeric order.

STAND-ALONE RESERVED WORDS

ALARM	DUART	PtCurrent	Saux1outch
BD1050BPS	ENB_CTSTX	RCV	Saux2inch
BD110BPS	ENB_ECHO	RCVD_BRK	Saux2ops
BD1200BPS	ENB_RX	RCV_RDY	Saux2outch
BD134BPS	ENB_RXINT	REM_LOOP	Sballoc
BD150BPS	ENB_RX_RTS	RESET_ERR	Sbaud_speeds
BD1800BPS	ENB_TX	RESET_MR	Sbfree
BD19200BPS	ENB_TXRTS	RESET_RECV	Sbinit
BD2000BPS	ENDAREA	RESET_TRANS	Sbitblt
BD200BPS	EVN_PAR	ROMTERM	Sblocked
BD2400BPS	FCast	RST_A_BCI	Sbotbits
BD300BPS	FIFOFULL	RVIDEO	Sbramputstr
BD38400BPS	FIPoint	Rect	Scbufs
BD4800BPS	FIRSTBIT	Rectangle	Sceil
BD50BPS	FIRectangle	Rpt	Scheckbram
BD600BPS	Flint	SC_clock	Scur
BD7200BPS	Fllong	SC_confirm	Scursallow
BD75BPS	FlpBitmap	SC_crosshair	Scursblt
BD9600BPS	FlpLayer	SC_cup	Scursinhibit
BLCK_ERR	FlpProc	SC_deadmouse	Scursinit
Bitmap	FlpText16	SC_move	Scursor
BonW	FlpTexture	SC_skull	Scursset
CBITS5	FlpWord	SC_sweep	Scursswitch
CBITS6	Flpchar	SC_target	Scurtabp
CBITS7	Flshort	SEND	Sdefont
CBITS8	Flvoid	SENDAREA	Sdellayer
CHAR_ERR	FRC_PAR	SLbox	Sdevtab
CPU	FRM_ERR	SLgrey	Sdisconnect
C_clock	F_CLR	SNAVAIL	Sdisplay
C_confirm	F_OR	SNLONGS	Sdiv
C_crosshair	F_STORE	SPECIAL	Sdownback
C_cup	F_XOR	SPt	Sdtr
C_deadmouse	Firm	SPtCurrent	Seqpt

STAND-ALONE RESERVED WORDS

C_move	HVERSION	SRect	Seqrect
C_skull	KBD	SRpt	SetupFlag
C_sweep	LASTBIT	SSetupFlag	Sexcep_int
C_target	LCL_LOOP	STOP_BRK	Sexcep_msg
Code	Lbox	STRT_BRK	Sexcep_norm
DACast	Lgrey	ST_background	Sexcep_proc
DADDR	MOUSE	ST_black	Sexcep_stack
DCDA	MPX	ST_checks	Sfloor
DCDB	Menu	ST_darkgrey	Sfree
DCast	Mouse	ST_grey	Sfreeall
DIBitmap	NAVAIL	ST_lightgrey	Sfreelist
DIFont	NLONGS	ST_white	Sgcalloc
DIPoint	NO_OP	Sabs	Sgcfree
DIS_RX	NO_PAR	Saciagetc	Sgcfreeall
DIS_TX	NRML_MOD	Saciainit	Sgcinit
DITex	NULL	Saciapaws	Sgetnum
DITex16	ODD_PAR	Saciapoke	Shostgetc
DIchar	ONEP000SB	Saciaputc	Shostops
DIint	ONEP563SB	Saciarxint	Shostputc
DIlong	ONEP625SB	Saciatrint	Sinset
DIpBitmap	ONEP688SB	Sacioctl	Sinterrupt
DIpLayer	ONEP750SB	Sadd	Sjline
DIpProc	ONEP813SB	Saddr	Sjlineto
DIpRectangle	ONEP875SB	Salloc	Sjmove
DIpText16	ONEP938SB	Sallocb	Sjmoveto
DIpTexture	ONES	Sallocendp	Sjpoint
DIpWord	OVR_RUN	Sallocinit	Sjrectf
DIpchar	PAR_ENB	Sallocstartp	Sjsegment
DIpint	PAR_ERR	Sauto1	Sjstring
DIplong	POLY_F	Sauto2	Sjstrwidth
DIshort	PSEND	Sauto4	Sjtexture
DTRA	Point	Saux1inch	Skbdchar
DTRB	Pt	Saux1ops	Skbdinit

STAND-ALONE RESERVED WORDS

Skbdrepeat	Sscroll	_2681	excep_msg
Skbdrpt	Ssegment	abs	excep_norm
Skbdstatus	Ssendbreak	aciagetc	excep_proc
Skgetc	Ssendchar	aciainit	excep_stack
Slayerop	Ssetbram	aciapaws	floor
Slback	Ssetup	aciapoke	free
Slbitblt	Ssopbaud	aciaputc	freeall
Slblt	Ssopoke	aciarxint	freelist
Slfront	Sspl0	aciatrint	gcalloc
Sload	Sspl1	acioctl	gcfree
Slogports	Sspl4	add	gcfreeall
Slpoint	Sspl5	addr	gcinit
Slrectf	Sspl6	alloc	getnum
Slsegment	Sspl7	allocb	hostgetc
Sltexture	Ssplx	allocendp	hostops
Smaxaddr	Sstdcurtab	allocinit	hostputc
Smenuhit	Sstopbreak	allocstartp	inset
Smouse	Sstopdisc	auto1	interrupt
Smul	Sstring	auto2	jline
Snap	Sstrlen	auto4	jlineto
Snewlayer	Sstrwidth	aux1inch	jmove
Snextlong	Ssub	aux1ops	jmoveto
Sobsclean	Stest32	aux1outch	jpoint
Sown	Stexture	aux2inch	jrectf
Spfedit	Stopbits	aux2ops	jsegment
Spfkey	Strdisable	aux2outch	jstring
Spiohint	Strenable	balloc	jstrwidth
Spoint	Ststdcd	baud_speeds	jtexture
Spolyf	Sublocked	bfree	kbdchar
Sprcvchar	Supfront	binit	kbdinit
Sprntops	Susercurtab	bitblt	kbdrepeat
Sprobe_io	Sver_str	blocked	kbdrpt
Spsendchar	Sversion	botbits	kbdstatus

STAND-ALONE RESERVED WORDS

Sptinpoly	Svitty	bramputstr	kgetc
Sptinrect	Swait	bttn	layerop
Sqclear	TVIDEO	bttn1	lback
Sqgetc	TWOP000SB	bttn12	lbitblt
Sqinit	T_background	bttn123	lblt
Sqputc	T_black	bttn13	lfront
Sqputstr	T_checks	bttn2	load
Sqsetbram	T_darkgrey	bttn23	logports
Sqsort	T_grey	bttn3	lpoint
Squeues	T_lightgrey	cbufs	lrectf
Sraddp	T_white	ceil	lsegment
Srcvchar	Texture	checkbram	ltexture
Srealalloc	Texture16	cur	maxaddr
Srealballoc	UWord	cursallow	menuhit
Srealgcalloc	VERSION	cursblt	mouse
Sreboot	WORDMASK	cursinhibit	mul
SrectXrect	WORDSHIFT	cursinit	muldiv
Srectclip	WORDSIZE	cursor	nap
Srectf	WonB	cursset	newlayer
Sremote	Word	cursswitch	nextlong
Sringbell	XMAX	curtabp	obsclean
Srsubp	XMOUSE	defont	own
Ssavecur	XMT_EMT	dellayer	pfedit
SsccAgetc	XMT_RDY	devtab	pfkey
SsccAioctl	YMAX	disconnect	piohint
SsccApoke	YMOUSE	display	point
SsccAputc	ZEROP563SB	div	polyf
SsccBgetc	ZEROP625SB	downback	prcvchar
SsccBioctl	ZEROP688SB	dtr	prntops
SsccBpoke	ZEROP750SB	duart	probe_io
SsccBputc	ZEROP813SB	eqpt	psendchar
Ssccinit	ZEROP875SB	eqrect	ptinpoly
Sscreenswap	ZEROP938SB	excep_int	ptinrect

STAND-ALONE RESERVED WORDS

qclear	remote	sendchar	strlen
qgetc	ringbell	setbram	strwidth
qinit	rsubp	setup	sub
qputc	savecur	sopbaud	test32
qputstr	sccAgetc	sopoke	texture
qsetbram	sccAioctl	spl0	topbits
qsort	sccApoke	spl1	trdisable
queues	sccAputc	spl4	trenable
raddp	sccBgetc	spl5	tstdcd
rcvchar	sccBioctl	spl6	ublocked
realalloc	sccBpoke	spl7	upfront
realballoc	sccBputc	splx	usercurtab
realgcalloc	sccinit	stdcurtab	ver_str
reboot	screenswap	stopbreak	version
rectXrect	scroll	stopdisc	vitty
rectclip	segment	string	wait
rectf	sendbreak		

LAYERS RESERVED WORDS

ALARM	FRM_ERR	PSEND	_2681
ALARMREQD	F_CLR	P_Array	abs
BD1050BPS	F_OR	Point	add
BD110BPS	F_STORE	Proc	addr
BD1200BPS	F_XOR	Pt	alarm
BD134BPS	GOTMOUSE	PtCurrent	alloc
BD150BPS	HVERSION	QUEUE_H	balloc
BD1800BPS	IPoint	RCV	bfree
BD19200BPS	IRectangle	RCVD_BRK	bitblt
BD2000BPS	I_Ref	RCV_RDY	botbits
BD200BPS	Iint	REM_LOOP	bttn
BD2400BPS	Ilong	RESET_ERR	bttn1
BD300BPS	IpBitmap	RESET_MR	bttn12
BD38400BPS	IpLayer	RESET_RECV	bttn123
BD4800BPS	IpProc	RESET_TRANS	bttn13
BD50BPS	IpTexture16	RESHAPED	bttn2
BD600BPS	IpWord	ROMTERM	bttn23
BD7200BPS	Ipchar	RST_A_BCI	bttn3
BD75BPS	Ishort	RUN	bttns
BD9600BPS	Ivoid	RVIDEO	button
BLCK_ERR	JERQPROC_H	Rect	button1
BLOCKED	Jdisplayp;	Rectangle	button12
BUSY	KBD	Rpt	button123
Bitmap	KBDLOCAL	SEND	button13
BonW	LASTBIT	SPECIAL	button2
CBITS5	LAYER_H	STKSIZ	button23
CBITS6	LCL_LOOP	STOP_BRK	button3
CBITS7	LOCKLAYER	STRT_BRK	canon
CBITS8	Layer	Sys	cbuf
CBSIZE	MINSTKSIZ	TASK_C	ceil
CHAR_ERR	MOUSE	TVIDEO	clipbttn
CONTROL	MOUSELOCAL	TWOP000SB	clist
CPU	MOVED	T_background	cursallow

LAYERS RESERVED WORDS

C_clock	MPX	T_black	cursinhibit
C_confirm	MPXINDIRECT	T_checks	cursset
C_crosshair	MPXPATCH	T_darkgrey	cursswitch
C_cup	MPXTERM	T_grey	display
C_deadmouse	MPX_H	T_lightgrey	div
C_move	Menu	T_white	duart
C_skull	Mouse	Texture	eqpt
C_sweep	NOPFEXPAND	Texture16	eqrect
C_target	NO_OP	UNBLOCKED	exit
Cast	NO_PAR	USER	floor
Code	NPROC	UWord	free
DADDR	NRML_MOD	VERSION	gcalloc
DCDA	NULL	WAKEUP	gcfree
DCDB	ODD_PAR	WORDMASK	getrect
DEMUX	ONEP000SB	WORDSHIFT	inset
DIS_RX	ONEP563SB	WORDSIZE	itox
DIS_TX	ONEP625SB	WonB	jline
DTRA	ONEP688SB	Word	jlineto
DTRB	ONEP750SB	XMAX	jmove
DUART	ONEP813SB	XMOUSE	jmoveto
D_Ref	ONEP875SB	XMT_EMT	jpoint
Drect	ONEP938SB	XMT_RDY	jrectf
ENB_CTSTX	ONES	YMAX	jsegment
ENB_ECHO	OVR_RUN	YMOUSE	jstring
ENB_RX	Obscured	ZEROP563SB	jstrwidth
ENB_RXINT	P	ZEROP625SB	jtexture
ENB_RX_RTS	PANDORA	ZEROP688SB	kbdchar
ENB_TX	PAR_ENB	ZEROP750SB	max
ENB_TXRTS	PAR_ERR	ZEROP813SB	menuhit
EVN_PAR	PIPETO	ZEROP875SB	min
FIFOFULL	POLY_F	ZEROP938SB	mouse
FIRSTBIT	PROCSTATES	ZOMBIE	mul
FRC_PAR	PROC_H	ZOMBOOT	muldiv

LAYERS RESERVED WORDS

nap	ptinpoly	ringbell	strlen
newproc	ptinrect	rsubp	strwidth
outline	ptr_fint	rtransform	sub
own	qsort	screenswap	sw
pcb	raddp	segment	texture
pfkey	rcvchar	sendchar	topbits
physical	realtime	sendnchars	transform
point	rectXrect	setnorun	udata
polyf	rectclip	setrun	unblock
proctab	rectf	sleep	version
psendchar	request	string	wait

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Your comments and suggestions are appreciated and will help us to provide the best documentation for your use.

1. How would you rate this document for COMPLETENESS? (Please Circle)

   Excellent Adequate Poor
   4 ----------3 ----------2 ----------1 ----------0

2. Identify any information that you feel should be included or removed.

   _____

   _____

3. How would you rate this document for ACCURACY of information? (Please Circle)

   Excellent Adequate Poor
   4 ----------3 ----------2 ----------1 ----------0

4. Specify page and nature of any error(s) found in this document.

   _____

   _____

5. How would you rate this document for ORGANIZATION of information? (Please Circle)

   Excellent Adequate Poor
   4 ----------3 ----------2 ----------1 ----------0

6. Describe any format or packaging problems you have experienced with this document.

   _____

   _____

7. Do you have any general comments or suggestions regarding this document?

   _____

   _____

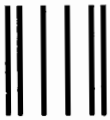
8. We would like to know a little about your background as a user of this document:

A. Your job function _____

B. Number of years experience with computer hardware: operation _____ maintenance _____

C. Number of years experience with computer software: user _____ programmer _____

Your Name _____ Phone No. _____
Company _____
Address _____
City & State _____ Zip Code _____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 1999 GREENSBORO, N.C.

POSTAGE WILL BE PAID BY ADDRESSEE

DOCUMENTATION SERVICES
2400 Reynolda Road
Winston-Salem, N.C. 27106-9989

Do Not Tear—Fold Here and Tape

Your comments and suggestions are appreciated and will help us to provide the best documentation for your use.

1. How would you rate this document for COMPLETENESS? (Please Circle)

Excellent		Adequate		Poor
4	3	2	1	0

2. Identify any information that you feel should be included or removed.

_____

_____

3. How would you rate this document for ACCURACY of information? (Please Circle)

Excellent		Adequate		Poor
4	3	2	1	0

4. Specify page and nature of any error(s) found in this document.

_____

_____

5. How would you rate this document for ORGANIZATION of information? (Please Circle)

Excellent		Adequate		Poor
4	3	2	1	0

6. Describe any format or packaging problems you have experienced with this document.

_____

_____

7. Do you have any general comments or suggestions regarding this document?

_____

_____

8. We would like to know a little about your background as a user of this document:

A. Your job function _____

B. Number of years experience with computer hardware: operation _____ maintenance _____

C. Number of years experience with computer software: user _____ programmer _____

Your Name _____ Phone No. _____

Company _____

Address _____

City & State _____ Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 1999 GREENSBORO, N C

POSTAGE WILL BE PAID BY ADDRESSEE

DOCUMENTATION SERVICES
2400 Reynolda Road
Winston-Salem, N.C. 27106-9989

Do Not Tear—Fold Here and Tape