INTERNAL DESCRIPTION

OF MIDAS

30 November 1979

by

Edward R. Fiala

Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA. 94304

Filed on: [Ivy]<DoradoDocs>MidasInternal.press

Sources on: [Ivy]<DoradoSource>MidasInternal.dm

This document describes the internal organization of Midas, so that anyone who wishes to do so can adapt Midas to a new hardware system.

TABLE OF CONTENTS

1. Introduction

This document is intended to serve as a guide for people who wish to adapt Midas for use as a debugger on a new hardware configuration. It is revised from time-to-time, but should not be relied on for total accuracy--see the sources to answer detailed questions.

2. How to Get Started

There are two Midas implementations which you might wish to use as a starting point: D0 and Dorado. Both implementations share a large body of machine-independent code, with D0 Midas not using some of the hardware checkout modules. D0 Midas interfaces to the hardware by exchanging messages with a kernel microprogram running on the D0; it can cause the D0 to boot itself, but its control over the hardware is then limited to whatever the kernel can do; the kernel program implements stop-and-go, insert/remove breakpoints, and register/memory read-write.

Dorado Midas controls the hardware directly, which allows a number of hardware checkout aids that would be impractical on the D0. Dorado Midas is more extensively developed, with numerous actions for setting the clock speed, doing parity-error scanning, hardware configuration control, and other features.

Breakpoints are simply implemented on Dorado, complicated on the D0 (because Dorado microinstructions have breakpoint bits, while D0 microinstructions don’t, so on D0, the broken instruction must be saved and restored). Microprocessor stop and go is complicated on both machines. Hardware reset is complicated on the Dorado, less so on the D0.

If you want a quick, primitive Midas implementation, you should probably start with D0 Midas; if you want a medium to fully developed debugging system, then your choice might be governed by whether Midas will control the target machine through a kernel program (in which case start with D0 Midas) or directly (in which case start with Dorado Midas); you may wish to obtain listings of both versions for comparison purposes.

The discussion below assumes that you start with the Dorado Midas; the instructions below would be similar for D0 Midas, for the most part replacing "Dorado" and "D1" by "D0" is would you would do.

1. Step 1 was to obtain this document ([Ivy]<DoradoDocs>MidasInternal.Press) and the user manual ([Ivy]<DoradoDocs>MidasManual.press); you should read the user manual through, skipping sections obviously specific to the Dorado implementation. Then you should read the first part of this document (stop when you get to the description of the external procedures); then return to these instructions.

2. Obtain a blank (i.e., useless) Alto disk, ether-boot the NetExec, and run NewOS.boot. Go through the long installation dialog selecting the "Erase a disk before installing" option. When that finishes, retrieve [Ivy]<DoradoSource>MidasDevelopmentDisk.Cm and execute it. This will setup your Alto disk with the files needed to prepare Dorado Midas. Note that your disk will have been loaded with .br files from MidasOtherBinaries.Dm but will not contain the sources for these; if you have to obtain these sources for some reason, they are on [Ivy]<DoradoSource>MidasOtherSources.Dm.

3. Use Empress to print D1Prin2.Bcpl, D1Reset.Bcpl, D1GoOvl.Bcpl, and D1LdrTest.Bcpl. You won't need these files, but they may be useful to you as examples, so get the listings now.

4. Delete D1Prin*.*, D1Sim*.*, D1Mic*.*, D1Config.Bcpl, D1Reset.Bcpl, D1Poke.Bcpl, D1Ti.Bcpl, D1hwcheck.Bcpl, D1mic*.*, D1instrs.D, and D1DMux.D. These source files for the Dorado Midas are totally uninteresting to you, except for the ones which you have already listed. The remaining files named D1*.* contain stuff which you will be modifying to adapt Midas to your new hardware system.

5. The disk prepared with the MidasDevelopmentDisk.Cm command file includes the runtime files for Dorado Midas. If you have not used Midas before, you might want to try out the Dorado Midas on your Alto to see how different features work--it will operate intelligibly without any hardware connected to your machine. After playing with Midas.Run, delete all of the files named *.Midas except for Midas.Midas and Special.Midas and delete Midas.Run, Midas.Syms, Midas.Bs, Midas.Fixups, Midas.Compare, Midas.RunProg, and Midas.Dtach; DO NOT DELETE Midas.Bcpl, Midas.Errors, or Midas.Programs.

6. Your disk will also have Neptune.Run, Chat.Run, Micro.Run, and Empress.Run, which can be deleted if space is tight.

7. Print *.Bcpl, *.Asm, *.D, *.Cm, *.Midas, *.mc, and Midas.Programs; make two notebooks sorted in the order of the files in MSources.Cm and D1Sources.Cm; you have already deleted most of the files named in D1Sources.Cm, so don't worry about the ones that don't exist. In each notebook you can conveniently group all *.Cm files into a single section; use another notebook section for *.Midas and Midas.Programs. You will gradually replace the files in D1Sources.Cm by your edited versions of these as you adapt Midas to your new hardware configuration.

8. There are only a few edits to the files in MSources.Cm required, as discussed in the "Hardware Interface" section later.

9. You should recompile the machine-independent part of Midas with @MComp.Cm when you have completed the edits for ValSize, TValSize, etc.

10. Edit the D1xx.Bcpl and D1xx.Asm files and convert these as appropriate for your hardware; comments in the listings should serve as a guide for how to do the editing. Replace the "D1" part of these file names with a good prefix for your hardware (e.g., "M68", "D0", etc. have been used for other implementations of Midas); write your edited versions of the D1*.* files onto files with appropriately chosen names. When you are happy with your edited versions, you can delete the D1*.* files (but probably retain the original listings in case some problem comes up later).

11. A good order for editing the files is as follows:
a. D1.d and D1regmem.d
b. D1i0.bcpl (use the Doradomc.Br microcode file until such time as you implement your own special microcode).
c. D1i1.bcpl
d. D1i2.bcpl
e. D1Tables.Asm (fill in each table with stuff for your registers and memories--all edits are straightforward; initially use RDatatoCSS and MDatatoCSS for all the prettyprint table entries; later

replace these with special prettyprint procedures when you code them).
f. In D1Asm.Asm edit CertifyAV and ConvertAV for your hardware; retain the error strings and the GetFrame thing near the beginning of the file; delete approximately everything else.
g. D1reg.Bcpl and D1mem.Bcpl (complicated edits to create the Get and Put procedures for your registers and memories).
h. D1res.Bcpl (may want to put some code in ReadAllRegs depending upon how you arrange the Get and Put interface to your hardware).
i. D1Go.Bcpl (complicated edits and the procedures are not designed very well, but can probably struggle through these eventually; may have to change MGo.Bcpl also--many changes were made in MGo.Bcpl for D0).
j. D1Debug.Bcpl (probably no edits required)
k. D1Field.Bcpl and D1Fieldasm.Asm (may want to rip this out by removing the entry for the field test in D1actions.Asm; otherwise easy edits)
l. D1Test.Bcpl D1TestAsm.Asm (easy edits, but might want to eliminate the Test, TestAll, continue-test, and continue-tall items from D1Actions.Asm altogether)
m. D1Load.Bcpl (easy edits).
n. D1PEScan.Bcpl (might want to rip this out unless you have memories that can be scanned for parity errors).
o. D1Actions.Asm--may want to remove some actions or add some; may want to change some of the flags in the table and in MCommon.D.
p. D1VM.Bcpl--probably want to retain FastSearch and RetrieveBlock procedures; VA/AA stuff may or may not be relevant to your hardware.

12. Replace the D1Sources.Cm, D1Binaries.Cm, D1Comp.Cm, D1DumpSources.CM, and D1DumpRun.Cm command files with the equivalents for your implementation.

14. Edit the D1Load.Cm command file as appropriate for your implementation; be careful not to change the ordering of the Init2, Load, and Test overlays; be sure to leave the init0 and init1 files in the correct place. There is some advantage in putting the most frequently accessed overlays immediately after Test because Midas will fill any left-over core space with the first few overlays.

15. Copy the relevant Builtin definitions and mimic the memory definitions on page 1 of D1DLang.Mc and mimic the address symbol definitions in Loader.Mc to create a micro source file which defines the memories in which you have pre-defined symbols. This will include copying the stuff for the MDATA memory (BITS-CHECKED, etc.) and the MADDR memory (LOOP-COUNT, etc.) from Loader.Mc; the other stuff in D1DLang.Mc and Loader.Mc is irrelevant. You will wind up with a source file that can be assembled with Micro to produce a .Mb (micro-binary) file which defines these addresses for loading by the Midas.Midas command file.

16. Edit the Midas.Midas command file (controls the initial display configuration) and Midas.Programs (defines command files).

17. Edit the D1DumpRun.Cm command file as appropriate for your Midas; it should contain the names of the *.midas files that you ordinarily dump as part of a Midas release.

18. Recompile the machine-dependent part of Midas with your equivalent of @D1Comp.Cm.

NOTES:

1. Avoid changing stuff in the machine-independent part of Midas because this will prevent you from incorporating new releases of Midas in the future without massaging the files you have edited. A particularly troublesome area in Midas is the "Go" stuff which you may be tempted to change because it is rather kludgey; if you think of a better way to do this, please talk with Fiala to see if this can be made better.

3. General Organization and Command Files

Midas is stored on [Ivy]<DoradoSource> in five dump files:

MidasSources.Dm	machine-independent source files
MidasDoradoSources.Dm	machine-dependent source files for Dorado Midas
MidasOtherBinaries.Dm	.Br and some other files needed by Midas that don't have sources in MidasSources.Dm (MDI.Br, LoadRAM.Br, Overlaysinit.Br, DoradoMC.Br, Gacha10.Br, MDirs.Br, DirScanA.Br, DiskStreamsScan.Br, KeyStreamsA.Br, and KeyStreamsB.Br)
MidasOtherSources.Dm	source files for MidasOtherBinaries.Dm and some other stuff not ordinarily needed during Midas development
DoradoMC.Dm	sources for special Alto microcode needed for Dorado Midas (also need the BCPLRuntime microcode package).

Midas also uses (from [Maxc1]<Alto>) Time.Dm (contains TimeConvA.Br, TimeConvB.Br, and TimeIO.Br), and LoadRam.Br. MidasOtherBinaries.Dm contains OverlaysInit.Br and MDI.Br, which are variations of standard packages that you should *not* replace by more recent releases. Gacha10.Br (in MidasOtherBinaries.Dm) is obtained by applying the program MakeFont (from MakeFont.Bcpl) to Gacha10.Al.

The DoradoMC.Br file, which contains special Alto microcode used by Midas in driving the Dorado, includes the BcplRuntime package; sources for DoradoMC.Br are in [Ivy]<DoradoSource>DoradoMC.Dm; you also need [Maxc1]<alto>PackMu.Run (?), Mu.Run (?), and ?? to do Alto microcode development.

The machine-independent sources contain four declaration files called MDecl.D, MCommon.D, MAsmCommon.D, and MMenus.d. MDecl.D and MMenus.d should *not* be used by any machine-dependent files (except that xxLoad.Bcpl may include MDecl.D). MCommon.d and MAsmCommon.d may be used by both machine-dependent and machine-independent sources.

Naming conventions are as follows: Midas sources all begin with "M"; machine-dependent sources with a distinctive sequence (e.g., "D1", "D0", or "M68"). The machine-independent command files are MSources.Cm, MBinaries.Cm, MComp.Cm, and MDumpSources.Cm; machine-dependent command files are D1Sources.Cm, D1Binaries.Cm, D1Comp.Cm, D1DumpSources.Cm, and D1Load.Cm (or "D0...", or "M68...").

4. Source File Organization

The one important convention in organizing the source files is that the linkage in the "external" statements should be expicitly shown, so that every externally-referenced static is preceded by a comment showing what file it is in. This allows references to be tracked down when changes are made or during debugging. See the machine-independent sources for examples of this and for other formatting conventions.

5. Picking Up New Midas Releases

After you have started developing Midas for a new hardware system, there may be new releases of the machine-independent software which you want to pick up. To do this you will load the new version of [Ivy]<DoradoSource>MidasSources.Dm and the single file D1Actions.Asm from MidasDoradoSources.Dm. However, before doing this, you will have to preserve any machine-independent files that you have modified (probably by renaming them), so that you can resolve your modifications against those in the new release.

You will probably have modified MCommon.D, maybe MAsmCommon.D, maybe MGo.Bcpl, and certainly xxActions.Asm. Since you have saved the listings of these files from the previous Midas release, you will be able to identify changes in the new release and can then merge your edits with those in the machine-dependent sources. Then recompile the machine-independent sources.

6. Storage Allocation

The Midas allocator is only used during initialization. It acquires all storage between the end of Midas code and the OS, Junta'ed to levKeyboard. From this all other storage needed is allocated.

All storage is allocated using either GetStorage(Size) or GetEvenStorage(Size). Blocks allocated by these routines do not have any header word and cannot be deallocated. Midas does *not* use GetFixed.

Storage is allocated beginning just below the OS and working down toward Midas code at lower addresses. This direction of allocation is desirable because it permits storage occupied by initialization procedures to be added to the storage area after execution without fragmentation.

The allocation is divided into three stages. During Init0 (in the MInit0.bcpl file) big blocks and blocks which can be initialized quickly are allocated. This is done prior to a Midas/I check in Init0B(). Storage allocated after the Midas/I check is preserved on a state file. This should be limited to small blocks which take a long time to initialize.

When Midas/I is not typed, this code is not executed and the storage area is instead restored by RestoreState(..) (in Midas.bcpl) from the state file. Presently, Midas/I takes about 10 seconds compared to about 3 seconds when Midas/I isn't typed. This varies somewhat according to placement of files on the disk and details of the machine-dependent code; these times are measured from the beginning of Midas initialization and do not include approximately 5 or 6 seconds consumed by the Alto Executive to roll in the Midas.run core image from the disk.

Note that two state files must be created by the Midas.Midas initialization command file: Midas.RunProg, from

which state is restored on a RunProg action, and Midas.Dtach, from which state is restored on a Dtach action. Other Midas implementations may not require any distinction between these two. The easiest method is to leave Midas unchanged and write the extra state file, even though it is unneeded; this will leave the machine-independent sources untouched.

If you want to eliminate the extraneous state file, Midas.Dtach, remove the SetupFPN calls for Midas.Dtach from MInit0.bcpl (and fix up the file name indices and the NFNames manifest); then remove the extra WriteState action from Midas.Midas; then replace the "(Dtach ? DtachFP,RunProgFP)" clause in the CmdRunProg procedure in MCmd.bcpl by "RunProgFP".

Final initialization is carried out by the Init2 overlay. Storage allocated by Init2() is not preserved on the state file. The final action of Init2() is to load overlays into any unused storage; these overlays get peeled away as a microprogram being loaded consumes the storage for symbols.

The first big block of storage allocated by Init0 is for display bit buffers. Presently, all but 9 buffers are also used as an OverlayZone when commands in overlays are executed. At these times the top part of the display is blank. The Alloc package in the OS is used to manage OverlayZone. Overlays are not written back on the disk when they are flushed.

7. Display Management

Midas has its own baroque display management stuff, handled in three levels. The lowest level consists of a pool of bit buffers, a table of line control blocks (An LCB consists of one DCB for the text and one for the leading between lines), and a vector of strings called ScreenTV. The screen size is determined by the statics ScreenHeight and ScreenWidth in the machine-dependent part of Midas.

An effort has been made to manage the display quickly without requiring much extra storage. At the same time, the algorithms avoid trashing the display by ensuring that bit buffers are not in use when they are grabbed for use in OverlayZone or for another screen line.

Two extra bit buffers exist. The number of extras is significant after a "Go" or "Test" action when many registers have changed. In this case, the algorithm, to avoid trashing the screen, must paint a new bit buffer for each changed line, then wait for the vertical interrupt to ensure that the old bit buffer is no longer in use, and finally assign the new bit buffer to the DCB and free up the old one. The number of extra bit buffers and the time required to convert a screen line into a bit buffer determines how many times it is necessary to wait for the vertical interrupt in this situation. It was empirically determined that two extra bit buffers worked well enough.

Strings pointed to by ScreenTV are painted into bit buffers by ClearAndScanConvert (in MAsm.Asm), called by GetFreeBitBuffer (MDisp.Bcpl).

Above the low-level display driver is the Region stuff in MRgn.Bcpl. Presently there are three kinds of regions: menus, the input text line, and comment lines. More about regions in a moment.

At the top level, text is sent to the display by stream operations (Wss, Wns, Puts, Resets, etc.) The streams first write into TextVec's. A TextVec is a one-character/word vector with word 0 containing the number of characters.

All menus are built by writing initially into the ItemV TextVec, as discussed later. MarkMenus moves the characters from ItemV into ScreenTV using PaintItem (hand-coded).

The input text line, command comment lines, and program line have separate TextVec's. When these are written via Puts or Resets, the DispDirty flag is set for the region. DispDirty causes PaintRgnLine to be called eventually (from DriverLoop's call to UpdateDisplay or by a direct call to UpdateDisplay), and that moves the text into ScreenTV.

The algorithms attempt to avoid unnecessary screen rebuilding. This is done as follows: First, UpdateMPDValues() in MMPRgn.Bcpl is called by the machine-dependent code, when it believes the displayed values should be reread from the hardware; the machine-independent part of Midas never calls this procedure. UpdateMPDValues() will call the MGetMemData or MGetRegData procedures in the machine-dependent code for every memory word or register currently displayed, update the value in the MPD structure for that item, and set a flag indicating that the display should be updated, if the new value is different from the old. MGetMemData and MGetRegData may return false, if, for some reason, the value cannot be obtained from the hardware at that time (e.g., when the machine is running, these procedures may return a false value).

The machine-dependent code can also call UpdateMPDValues(true) which is like UpdateMPDValues() except that only those displayed items marked with the AlwaysUpdate attribute (from the MEMCON or REGCON table) will be updated. This variation is used for items that are dynamically updated on the display, such as the UPTIME register on Dorado Midas.

Next, MPDEveryTime(..) in MMPRgn.Bcpl is called by the Midas DriverLoop procedure. MPDEveryTime will rebuild ScreenTV for every MPD item that has been marked by UpdateMPDValues (and there are several other reasons for rebuilding the text vectors). It sets the ScreenLineDirty vector to true for each screen line that has changed. Only when this flag is true is a new bit buffer constructed.

When the display is off during command files, display rebuilding is deferred until the display is turned on again.

8. Symbol Table

The Midas symbol table is an alphabetically sorted two-level symbol table consisting of a 256-word HeadBlock at the top level and one or more 1024-word Blocks in core. The final allocation by Init2 assigns unused storage to overlays. The overlays are peeled off and more Blocks made as symbol table storage is needed for a microprogram.

When the symbol table size overflows resident core Blocks, it is swapped off Swatee (It may be desirable to replace Swatee by Midas.SymTab during initial debugging, and a simple edit to MInit0.Bcpl will make this substitution.). The record file (Swatee), managed by GetBlock, PutBlock, etc. procedures in MSym.Bcpl and MSymOv.Bcpl, is also used for virtual memory mapping information and for the FastSearch stuff discussed below.

HeadBlock and regular Blocks have identical structure: a header defined by the BH structure in MDecl.D, followed by a table of block-relative pointers growing upward from the header; symbol records grow downward from the end of the block.

Symbol records consist of a bcpl string followed by the value part of the record. In the HeadBlock,

all values are one-word long--the index of that page on Swatee (called BlockAddr). The HeadBlock contains one entry for the smallest symbol on each regular Block. In regular Blocks, the value part of the record is one of the forms described by the Symb structure in MDecl.D.

Code for managing the symbol table is in MSym.Bcpl (resident) and MSymOv.Bcpl (Load overlay). New symbols are entered only during initialization and loading, so these procedures can be in an overlay. EvalAText looks up symbols and UpdateRcd enters new ones. SearchBlocks scans for the nearest less-equal symbol to a particular value. SearchBlocks is called after SS and Go to show the symbolic equivalent of the address at which the microprogram stopped (SearchBlocks is also used in many other situations). It takes about .07 seconds to scan 1000 symbols with SearchBlocks plus about .15 seconds for each block read from the disk.

Large programs can result in the symbol table being spread over 20 or more symbol blocks, and SearchBlocks becomes slow. For this reason, SearchBlocks calls a procedure FastSearch(..) before scanning the blocks. The CleanUpBlocks() procedure in D1Load.Bcpl shows how inverted tables can be built for faster searching, and the FastSearch(..) and RetrieveBlock(..) procedures in D1VM.Bcpl show how this is handled. You may be able to incorporate the Dorado FastSearch code almost verbatim.

The symbol table is built as follows: First, Init1() enters symbols for the register and memory names. Next, the Midas.Midas command file loads some address symbols for LOW-ADDRESS, HIGH-ADDRESS, BITS-CHECKED, etc.--symbols in the fake MDATA and MADDR memories used by the Test and TestAll actions. The machine-dependent code may also enter a number of other symbols into the symbol table during Init1. Note that these address definitions are defined in an ordinary microprogram that is loaded during execution of the Midas.Midas command file.

Next, if the user loads a microprogram, address symbols in that microprogram are entered. Since Micro alphabetically orders symbols, UpdateRcd (in MSymOv.Bcpl) first compares a new symbol against the one last inserted and the one after that. If the new symbol is between these two, it is immediately inserted. Otherwise, a binary search is done on the HeadBlock to find the Block containing the new symbol, then another binary search in that Block to find the position.

If the new symbol won't fit on the Block selected, BlockSplit() is called. BlockSplit creates a new symbol Block, using unused core if possible, else peeling overlays until there is space for a Block, else writing a dirty block onto Swatee and reclaiming that space. The symbols on the Block that overflowed are divided between the old and new blocks.

BlockSplit is optimized for symbol insertions in alphabetical order into a table *nearly empty* at the beginning of the load. The division point leaves the original block at least as full as the BlockBreak parameter. If the new symbol is above BlockBreak, all symbols below it will be left in the original Block. Hence, the number of symbols above the division point is always less-equal to the number in the symbol table at the beginning of the load. If this number is small compared with the number that fit in a Block, then Blocks wind up nearly full, and any Blocks written on the disk will not have to be reread during loading. If the symbol that caused overflow is below BlockBreak, then it will be inserted on the original block and it may be necessary to read the split-off block from the disk later. For these reasons, it is desirable for the number of symbols at the onset of a load to be small (This is unfortunately not the case on Dorado).

9. Actions

All work done by Midas is initated by Actions. Actions may be invoked either by clicking the mouse over a menu item, by naming the Action in a command file, or by typing the command character(s) for the Action on the keyboard.

All permanently-defined actions are in the xxActions.Asm file. This consists of a big table with 5 words per action, interpreted according to the Action structure in MDecl.D. Many actions in the command menu use CreateAction(..) to create temporary actions during execution; the actions created this way are written into the unused area at the end of the ActionBlock table. These are destroyed at exit by QuitCmdOverlay(..).

The way the parts of an Action are used is discussed later in the description of the CreateAction(..) procedure. The MenuMChange(..) procedure in MMenu.Bcpl and the DoAction(..), DoTextAction(..), CreateAction(..), and ForgetTemporaryActions() procedures in MMenu.Bcpl are the ones relevant to actions.

In the ActionBlock in xxActions.Asm is a sequence of actions that appear in the primary command menu. These are put up by FormCmdmenuText() which calls ShowActions(..) in MCmd.Bcpl. Flags in the last word of each Action control when it appears in the menu, and you may want to modify the flags and the code in FormCmdmenuText(..) for a particular machine.

Actions which put up alternate menus are normally invoked by StartCmdOverlay in the lvProc entry of the Action, and the procedure that returns a procedure for forming the alternate menu is the Arg word in the Action. StartCmdOverlay(..) does ErrorProtect(lvProc, ..). A good example of a complicated action that uses an alternate command menu is in D1Reset.Bcpl (which you listed).

10. Menus

Every name-value area on the display is a Menu, and the command area at the bottom of the screen is also a Menu. Each menu is described by an MDFS structure as given in MDecl.D.

Name-value menus are defined in the machine-dependent InitMPInterface() procedure called by Init1(). InitMPInterface() calls NewMPDispBlock(..) which in turn calls MakeMenuRegion(..) to carry out the definition. The command menu is defined by a call on MakeMenuRegion in MInit1.Bcpl.

DriverLoop(), the main loop of Midas, reads mouse information each time through, detecting changes in position or mouse buttons. When the mouse migrates across region boundaries, deSelectMenu(..) is called to make a deselected menu region be black-on-white again. MenuMChange(..) is called when the mouse buttons or character position change in a menu region. MenuMChange(..) in turn calls the mouse-change and Action procedures defined in the Action table entry for the selected Action.

In addition, deSelectMenu() calls MPDdeSelect(..) for name-value menus.

Menu formation delimits strings as menu items each paired with an Action procedure called when a mouse button is released while that item is selected. MPDEveryTime(..) in MMPrgn.Bcpl loops over all name-value menus (MPD's), forming these menus with FormMenu(FormMPDmenu,MPD).

FormMenu initializes the menu region to empty and ItemStream to empty then calls FormMPDmenu(MPD). This routine puts up sequences of characters and calls MarkMenus(Action) to pair that sequence of characters with an Action. MarkMenus reinitializes ItemStream to empty. This mess winds up with the SizeVec and ProcVec vectors in the MDFS structure for that menu region describing the correspondence between characters on the display and Actions to carry out.

A similar sequence of procedure calls is used to form the command menu, but the composite procedure FormCmdMenu() is used to do this. Also, the WsMarkA(Action) procedure is used to do Wss(ItemStream,action-name); MarkMenus(Action); Puts(ItemStream,$); MarkMenus(0).

11. Every Time List

Midas uses a table of procedure names and arguments called the EveryTimeList for things that must be done every time through DriverLoop. MPDEveryTime is put on this list by Init1(); MPDEveryTime calls HWEveryTime to do any machine-dependent stuff (HWEveryTime will be a no-op procedure on most Midas implementations). Indefinite Action routines, such as the one for Go, add procedures to check for termination. The TimeOut action adds a procedure.

EveryTimeList is managed by AddToEveryTimeList(Proc,Arg) which returns a pointer that can be passed to RemoveFromEveryTimeList().

12. Files and sysZone

Midas initialization creates a stack (4400_8 words) and sysZone (6400_8 words); these sizes are controlled by parameters in Midas.Bcpl. Initialization also creates all temporary files needed by Midas and builds file pointers (FP's) for these and for all of the command and other files enumerated in Midas.Programs; the FP's allow the files to be opened quickly, improving response time.

Open files use storage in sysZone consisting of a one page buffer and a disk stream structure; this requires approximately 450_8 words per open file. In addition, OS directory management procedures use about 1200_8 words for buffering sysDir (the system directory) when files are opened for which FP's are not given. During normal operation of Midas, it is possible for the following files to be open simultaneously:

One command file;
One file on which commands are being written (WrtCmds action);
One output file on which text is being written (OpenOutput action);
Two files during Ld, Dump, or Cmpr actions.

This means that the worst case storage requirement, ignoring fragmentation of sysZone, is about 4100_8 words; it was experimentally determined that a minimum sysZone size of about 5000_8 words is needed. Dorado Midas uses a 6400_8-word sysZone because of the extra storage needs of its simulator (StartLargeOverlay action). It would be possible for the first three of the five files mentioned above to be opened and closed in an unfavorable order such that sysZone becomes fragmented, and if this were to happen a sysZone somewhat larger than 5000_8 words would be needed, but in practice this has never happened.

OpenFile also consumes considerable stack space, and it has been experimentally determined that a stack size of 4400_8 words is not much more than the minimum needed by Midas.

Note that Midas Junta's to levStreams; the OS keyboard module is loaded with the Midas resident, and the OS directory module is loaded as a Midas overlay. If Midas.Programs and Midas.UserPrograms are setup correctly, it is seldom necessary to swap in the directory module because FP's will exist for the files normally touched by users.

13. Errors and Overlays

Impossible conditions detected by Midas, that should properly result in a crash of the Midas program, result in the MidasSwat(ErrNo,A1, ..., A5) procedure being called, where ErrNo is the number of a string in the Midas.Errors file. MidasSwat is discussed in the Midas.Bcpl section later.

Other error conditions resulting from hardware malfunctions or user mistakes, result in an error diagnostic being printed by Midas and unwinding of the stack as discussed below.

All Action procedures are called via the ErrorProtect(lvProc,A1,A2,...) procedure. ErrorProtect saves the stack position in AbortLabel and AbortFrame and establishes an errorset to which calls to DisplayError(..), ErrorExit(..), or ErrorAbort(..) come back. In addition, iff either there was no OverlayZone or OverlayZone was empty at the time ErrorProtect was called, it will call KillOverlays()

before returning. In other words, the errorset established by ErrorProtect will clean up everything except the display on/off condition before returning.

OverlayFaultProc in MOverlay.Bcpl intercepts calls to not-in-core procedures and ReadOverlay(..) assigns storage for the overlay, reads it from the disk, and fixes up the procedure statics. Unless OverlayZone already exists, ReadOverlay converts all name-value bit buffers into an OverlayZone. For the present Dorado Midas, with 20 lines of name-value buffers, OverlayZone winds up at $20{,}642_8$ words, the size limit for overlays.

On Dorado Midas, the hardware simulator required more storage than this, so StartLargeOverlay in MCmd.Bcpl and StartBigOvlWithAddr in MGo.Bcpl were added. These procedures scrounge an additional 5100_8 words from sysZone and 3300_8 words from the stack and are available for command overlays that are too large to use StartCmdOverlay and StartWithAddr.

On other Midas implementations, the Load overlay will probably impose the largest storage requirement. When the FP for a file being loaded is unknown, this can result in the Load and Directory overlays being simultaneously in core together with the Load module's read buffer (presently 3400_8 words), which uses up nearly all of OverlayZone.

Unless called by ErrorProtect, the non-resident procedure will remain in OverlayZone until KillOverlays() is called. If procedures in different overlays are called, subsequent calls will also use OverlayZone.

Most overlays can't know whether they are running in OverlayZone or in left-over space between Midas resident and symbol Blocks (in which case no OverlayZone exists). In this situation, only sysZone is available for storage (6400_8 words). However, not all sysZone space is necessarily available, because of open file buffers discussed in the previous section.

A special exception is made for the Init2, Load, and Test overlays, which *must be the first three overlays in the xxLoad.Cm command file*. These are never preloaded and will always be in OverlayZone when running. The Init2, Load, and Test overlays are exempted from preloading by code in MInit2.Bcpl.

When KillOverlays() is called, all overlays in OverlayZone are flushed, and, if the display is on, OverlayZone is destroyed and the display rebuilt. If the display is off, KillOverlays() flushes the overlays, but the display is not rebuilt until the display is turned on again.

14. Hardware Interface

In adapting Midas to a new machine, the only changes required in the machine-independent part are as follows:

> The ValSize parameter in MAsmCommon.D should be set to the number of 16-bit Alto words required to hold the largest register or memory value, currently 4. ValSize must not be made smaller than 2. TValSize should be set to the number of 16-bit Alto words required to hold the largest testable register or memory value, currently 3. If ValSize or TValSize are changed, the machine-independent part of Midas must be recompiled.

D1Actions.Asm contains a 5-word entry for each permanently-defined Action. This includes both machine-independent and dependent Actions. This file has to be edited appropriately and should be renamed xxActions.Asm (where xx = D1, D0, M68, ...).

You may want to change some of the flags in the action table entries. FormCmdmenuText(..) in MCmd.Bcpl has to be edited if flags controlling the command menu are changed.

You will want to replace some of the CantContinue manifest constants in MCommon.D with ones appropriate for your hardware.

You will want to delete the DWatch, MCTimeOut, MIRlhPE, MIRrhPE, and possibly the CheckStoppedGlitches statics and the associated two-word vectors in MData.Asm; you may wish to add some other items to this table (which is the MADDR memory) for your machine or to the MDATA memory.

You will modify the Midas.Midas command file.

Replace the error strings above 2000 in Midas.Errors by ones for your implementation.

The machine-independent modules are supposed to manage all interaction between the user and Midas. They handle mouse and keyboard input, symbol table, menu formation, overlay management, etc. Machine-dependent subroutines are called to carry out specific interactions with the hardware.

Generally, machine-dependent procedures implement specific functions required by the machine-independent section; they do not call any complicated procedures in the machine-independent section.

In other words, there are only a few procedures in the machine-independent section ever called from machine-dependent modules.

They are as follows (discussed later):

Initialization procedures:
NewMPDispBlock, GetStorage, GetEvenStorage, GetZStorage, GetHStorage

Utility procedures:
ErrorProtect, DummyCall, DummyCall1, MBlock, MoveUp, GetField, PutField, MoveLongField, OddParity, VUsc, SelfRel, Mag, LCycle, DoubleNeg, DoubleAdd

Print stuff on streams:
Wss, Puts, Wns, DWns, DatatoStream, SearchBlocks
WssCSS, PutsCSS, ResetsCSS, WnsCSS, WnsCSSD
WssCS1, PutsCS1, ResetsCS1, WnsCS1, WnsCS1D
NWss, NWss1, PrinV0, PrinV1

Manage display:
FormCmdMenu, UpdateMPDValues, SetDisplay, Blink, UpdateDisplay

Convert input text:
CallWithAddr, StartWithAddr, CollectLDR

Manage EveryTimeList:
AddToEveryTimeList, RemoveFromEveryTimeList

Manage Actions:

CreateAction, WsMarkA, WssMAct, WssMark, MarkMenus, SetAbort

Confirmation and termination:
ConfirmAct, DisplayError, ErrorExit, ErrorAbort, QuitCmdOverlay

Numeric input/output:
SimpleTexttoDVec, DWns

Symbol table lookup and parsing:
SearchBlocks, EvalAtext, KindOfChar, ChkToken, SkipBlankToken, SymbKeyComp

Data patterns for testing:
GetPattern, SetupPattern, ShowPattern, SaveDGen, RestoreDGen, ContinueGen, CheckData, NextData, ErrorStop

Hardware interface procedures use the following data types as arguments and values:

DVec	Hardware data, multi-word, packed left-justified as in Micro format
AVec	Double-precision memory address vector
MemX	Index into memory tables
RegX	Index into register tables

Machine-independent files should be provided equivalent to the following:

D1i0.Bcpl	Defines the InitHardware1() procedure called by Init0(). This should load special Alto microcode (if any) and allocate big blocks of storage that do not have to be saved on the state file. The DoradoMC.Br microcode can be used for implementations that do not need their own special microcode; this contains the bcplruntime microcode which will speed execution (and some other microcode useful only on Dorado but harmless if not used).
D1i1.Bcpl	Defines InitMPInterface() called by Init1() and the zrel statics ScreenHeight and ScreenWidth. ScreenHeight contains 12+the number of lines in the name-value area. InitMPInterface() is only called for Midas/I initialization. It should make calls on NewMPDispBlock to define the name-value areas. It should call SaveStatics(..) for any zrel statics initialized.
D1i2.Bcpl	Defines InitHardware() called by Init2(). InitHardware() should do whatever initialization is necessary to ensure that GetRegData, GetMemData, PutRegData, and PutMemData, discussed below, will deliver accurate values. On the Dorado implementation of Midas, there is also code for determining the serial numbers of machines that are accessible to Midas and for selecting among these with a menu.
D1mem.Bcpl	Resident code defining MGetMemData, GetMemData, MPutMemData, and PutMemData.
D1reg.Bcpl	Resident code defining MGetRegData, GetRegData, MPutRegData, and PutRegData.
D1res.Bcpl	Resident code defining FormHWMenu, HWShowAddr, HWAlwaysUpdate, FinishHardware, DetachHardware, and HWEveryTime is discussed below. FormHWMenu builds conditional parts of the command menu.

	FinishHardware is called before exit from Midas; it should do a silent boot (if special Alto microcode requires this), and do any other cleanup. DetachHardware is called before detaching Midas from one machine and connecting it to another.
D1Tables.Asm	Resident tables defining the machine.
D1Go.Bcpl	Resident procedures called from MGo.Bcpl, MRGo.Bcpl, and MGoOvl.Bcpl. Defines SetupIMA(..), ContinueProgram(..), Stop(), CheckStopped(), MStopped(..), and OneStep(..). See sample code. On Dorado Midas, CheckStopped(..), ContinueProgram(..), and Stop() are hand-coded and appear in D1Asm.Asm rather than in D1Go.Bcpl.
D1Brkp.Bcpl	Defines BreakIML(..) called by InsertBreak(..) and RemoveBreak(..) in MBrkp.Bcpl.
D1Load.Bcpl	Part of the Load overlay. Should define PrepareLoad(SymOnly), RestoreAfterLoad(), PutMDSymOnly(..), and CleanUpBlocks() as discussed later.
D1Debug.Bcpl	Defines QuitTest(..) for Debug overlay.
D1Field.Bcpl	Defines FieldLoop, SetupFieldTest, and ModifyField for Field overlay.
D1FieldAsm.Asm	Defines tables needed by the Field overlay (mimic example faithfully).
D1Test.Bcpl	Defines ValidateTest(..) and additional test procedures for the Test overlay.
D1TestAsm.Asm	Defines tables used by the Test overlay.
D1PEScan.Bcpl	Defines ScanForPE(..) for the PEScan overlay.
D1VM.Bcpl	Defines FastSearch(..) called by SearchBlocks(..) and other procedures for the VM.

In D1Tables.Asm the following tables should be defined:

@MEMNAM	Memory names
@MEMWID	Memory widths
@MEMLEN	Double precision memory lengths
@MEMFORMS	Memory print format vectors for MTexttoData
AltMForms	Memory alternate print routines
AltMInput	Memory alternate typein routines
@MEMCON	Memory control bits
@NMEMS	Number of memories in the tables
@REGNAM	Register names
@REGWID	Register widths
@REGFORMS	Register print format vectors for RTexttoData

AltRForms	Register alternate print routines
AltRInput	Register alternate typein routines
@REGCON	Register control bits
@NREGS	Number of registers in the tables.

The @'s indicate zrel statics. Each memory table is NMEMS words long (except MEMLEN, which is 2*NMEMS long), and each register table NREGS words long. The MEMNAM, REGNAM, MEMFORMS, and REGFORMS tables contain self-relative pointers to the name strings or format vectors. These self-relative pointers are converted to real pointers by Init0(). See D1Tables.Asm file for examples of how these tables are setup. The order of the tables must be preserved faithfully because of the length error checks in MInit0.Bcpl.

The MEMFORMS and REGFORMS tables control printing/typein of values in the name-value displays. If the table entry contains a 0, the value is printed as a single octal number. Otherwise, the table points at a vector interpreted as follows:

If word 0 is positive (it should always be positive unless the register or memory has extensions) then

word -2:	lv Procedure for symbolic display mode (0 if none)
word -1:	MemX used in SearchBlocks display mode (-1 if none)
word 0:	number of groups for numeric display
word 1:	first bit of group 0
word 2:	number of bits in group 0
words (3,4), etc. up to (2n,2n+1):	more groups

If word 0 is negative, then it counts the number of extensions for the register or memory, each of which contains its own format vector in the above format, and the format table entry points to a structure as follows:

word 0:	- n (number of pointers)
words 1 to n:	self-relative pointers to vectors interpreted as above

Each of the groups is printed as an octal number followed by a blank. The format vector also controls evaluation of keyboard input for change-value actions. Keyboard input must be divided into groups bearing a one-for-one correspondence to the printout format.

Each group in formatted input or the single group for unformatted input may consist of an octal number, negative octal number, or address/memory name symbol +/- offset. If the input value won't fit in the register, the left-most bits of input are truncated. When the keyboard input cannot be evaluated in one of these ways, the routine named in the AltMInput or AltRInput table for the memory/register is called. Again, see D1Tables.Asm for examples. If this routine cannot evaluate the input, then it gives an error.

AltMForms and AltRForms contain pointers to procedures that pretty-print values on the comment lines. The pretty-print procedures are called as ErrorProtect(AltMForms!X,X,DVec,AVec,Radix+(Extension lshift 8)), where X is MemX or RegX, AVec is only relevant for memories, and Extension is non-zero only for multi-row items such as ROW on Dorado Midas. They are supposed to pretty-print the data in DVec on CmdCommentStream

and/or CmdCS1 (which have already been reset) in some meaningful way.

The MEMCON and REGCON tables contain for each memory/register an assortment of bits that determine how it should be handled by various Actions. The bits are defined in the MRType structure in MCommon.D. Included in the MEMCON/REGCON entry for each register and memory are the default printout mode and default radix that control input/output of values in the name-value menu area. The print out modes in the main display may be defaulted to either numeric, search-blocks, or symbolic.

Octal, decimal, and hexadecimal radices are implemented, with the overall default radix (8, 10, or 16) in the static DefRadix in D1I0.Bcpl overruled by the radix declared for each register and memory in MEMCON/REGCON.

In xxTestAsm.Asm the following tables should be defined:

MEMMask	Memory mask DVecs for Test and Test-All actions
MEMAnalyze	Memory lv Procedure for test failure analysis
MEMTST	Memory test conditions: pointers to statics that are true or false or to procedures that return true if the test is possible or false if impossible
MEMOne	Memory DVecs of value 1
REGMask	Register mask DVecs
REGAnalyze	Register lv Procedure for failure analysis
REGTST	Register test conditions
REGOne	Register DVecs of value 1
OtherNAM	Names of extra tests
OtherWID	Extra test widths
OtherProc	Extra test procedures
OtherPrint	Procedures to setup and print what the test is
OtherMask	Pointers to data-compare mask vectors
OtherTST	Extra test conditions
OtherOne	Extra test DVecs of value 1

15. Debug Overlay

The Debug overlay contains all of the pattern generation and data checking procedures used by other overlays during hardware checkout. This overlay is used on Dorado by the Field, LDRtest, Test, and SimTest overlays.

Data generation for testing uses a data pattern selected from a subsidiary menu as follows:

ZEROES	all 0’s data
ONES	all 1’s data
CYC1	left-cycled 1 in DVec of zeroes
CYC0	left-cycled 0 in DVec of ones
RANDOM	random numbers
SEQUENTIAL	sequential numbers starting at 0
SHOULD-BE	constant pattern equal to value in ShouldBe

ALTZO	alternating all-zeroes and all-ones
ALT-SHOULD-BE	alternates ShouldBe with its complement

Successive patterns are generated by the NextData() subroutine in MDebugAsm.Asm. CheckData() is used to check the results. SaveDGen() and RestoreDGen() subroutines are used to save the state of the pattern generator and restore it.

NextData(), CheckData(), and the memory tests use the following vectors of size TValSize (which are in the MDATA memory) defined in MData.Asm:

@BitsChecked	bits which should compare equal
@ShouldBe	holds data pattern generated by NextData
@DataWas	value read from hardware
BitsPicked	union of bits picked during test failures
BitsDropped	union of bits dropped during test failures

They use the following vectors of size 2 (which are in the MADDR memory), also defined in MData.Asm:

@LowAddress	lowest memory address tested
@HighAddress	highest memory address tested
@CurrentAddress	address last tested
@AddrIncrement	distance between consecutive addresses tested
AddrIntersect	intersection of addresses in which failures occurred during memory testing
AddrUnion	union of addresses in which failures occurred during memory testing.
@LoopCount	count of successful iterations prior to failure or to abort by ^C or mouse
TestFailures	count of test failures

The memory tables in xxTables.Asm should define the MDATA and MADDR memories, as in the sources. The Put and Get routines should, for this memory, move data to/from these tables and the DVecs.

16. Test Overlay

DataTest() displays a menu of all testable register and memory names. The MEMTST and REGTST tables determine whether or not a particular memory/register appears in this menu; these tables contain pointers to statics that are 0 (no test), -1 (always testable), or other (procedure that is evaluated to return 0 or -1). For registers PutRegData and GetRegData carry out the test. For memories, PutMemData is used to write consecutively all words in the address range being tested; then the pattern generator is restored and the words are read with GetMemData and checked. MEMMask and REGMask tables contain pointers to DVec's that mask the data being checked; the DVec's may be modified by the procedure in MEMTST/REGTST. Errors are reported when (data written xor data read) & BITS-CHECKED & Mask is non-zero.

The Midas manual discusses the way in which the Test and TestAll actions work.

For Test, the value originally in BITS-CHECKED is masked initially by a vector of ones of size equal to the register/memory width. Then if the RTAllTab/MTallTab entry (in xxTestAsm.Asm) for the register/memory is non-zero, it is a self-relative pointer to a vector of length ValSize in DVec format; this vector is and'ed with BITS-CHECKED.

17. Field Overlay

FieldTest() allows any microinstruction field to be repeatedly executed from the Alto and modified under mouse control for scope loops. Values for all fields except the one selected are taken from the NOP (no-operation) microinstruction. The instruction is repeatedly executed through the hardware interface. When the loop is stopped, the final instruction tested is left in TSTINS.

18. PEScan Overlay

The PE Scan overlay selectively scans memories for parity errors, allowing various options taken from a subsidiary menu. The memories which contain checkable parity and should appear in this menu are indicated by a bit in the MEMCON table.

The machine-independent code calls the machine-dependent ScanForPE procedure to do the actual scan and provides a ReportPE procedure by which errors are reported.

External Procedures and Statics by File

MIDAS.BCPL (resident)

NMidas(LayoutVector,userParams,CFA)

Starting address of Midas. Calls StartTimer to begin timing the initialization, then calls Init0 in MInit0.Bcpl to begin initialization.

During initialization many timing statics in Midas.Bcpl are filled in to reveal where time is being spent during initialization; they can be looked at with Swat.

InitRes(SlashI)

Entered by a GotoLabel from the Init0A procedure in MInit0.Bcpl. SlashI is true on Midas/i initialization, else false.

On Midas/i, the Init1 procedure is called to do a full initialization; then the Midas.Midas command file is executed to load address symbols and setup two initial display configurations: the display configurations are written on the Midas.RunProg and Midas.Dtach state files under control of the command file. The WriteState action writes the state files.

When SlashI is false, the initialization that would have been done by Init1, etc. is instead carried out by rolling in the Midas.Dtach state file with RestoreState.

Init2 is called to do the final initialization and parse Com.Cm. It calls InitHardware to do the machine-dependent part of this initialization. The command-file named in Com.Cm or on the command line for RunProg, if any, is opened and the stream returned; InitRes then calls CmdDoRC to execute the command file.

RestoreState is also called on RunProg and Dtach to reinitialize Midas prior to executing a command file. RestoreState exits with GotoLabel to the "Resume" tag in InitRes.

Command file execution jumps to the "BegCF" label in InitRes which pushes the current command file on the command file stack and begins execution of the new one; when the new command file finishes and returns, InitRes pops the command file stack, reopens and positions the previous command file and resumes it.

When the command file stack becomes empty, DriverLoop is called (the main loop of Midas) and runs until exit.

StartTimer()

Stores the timer in TimeStart and sets the TimerGoing static true. StartTimer() is called at the onset of Midas initialization, and at the onset of the first command file (in a possibly nested sequence).

ElapsedTime(lv Location)

Put the low-order elapsed time since TimeStart into Location; return the high-order elapsed time (which is 0 for times less than about 65 sec) as the result.

PrintTime(V)

Prints the double-precision vector V as an elapsed time on TimeStream rounded to 100ths of a second.

MidasFinish()

Procedure invoked by bcpl finish prior to exiting Midas. It calls FinishHardware() to do the machine-dependent part of the finish.

MidasSwat(ErrNo,p1,p2,p3,p4,p5)

Does a #77403 swat call with ErrNo as the number of a message in the file "Midas.Errors"; p1 to p5 are parameters used by the Swat error-printing procedure.

lvFinishProc

holds the value of rv lvUserFinishProc when Midas entered (written by Init2, read by MidasFinish).

Initialized

True if Midas initialization has completed (signals that the Get... and Put... procedures will deliver valid data; set true immediately after InitHardware in xxI2.Bcpl has finished initializing the hardware).

BBblockSize

The size of the display bit buffers, a complicated function of the screen size and the font computed by Init0. The value of this static is needed before its value is computed. Hence, the value is compiled into Midas and used during early initialization; later the compiled-in value is checked against the computed value, and Swat is called if there is disagreement. The user can fix the compiled-in value and try again, if this happens. This will happen when ScreenHeight or ScreenWidth changes.

AvailBlockSize

The size of the subpart of BBblockSize used to form an OverlayZone.

TopFrame

Points at the top stack frame which is the InitRes procedure; used by GotoLabel calls on RunProg, Dtach, and RdCmds actions.

SysZoneSize

Size of sysZone, used by MInit0.Bcpl.

StackSize

Size of stack, used by MInit0.Bcpl.

StateFileSize

Size of Midas.Dtach and Midas.RunProg, filled in for statistical purposes.

MINIT0.BCPL (first initialization--code flushed after execution)

Init0(Layout,userParams,CFA)

Puts information in the Layout vector passed by the Exec into FirstStatic, LastStatic, NStatics, RelocTable, Switch (/I), and MidasCFA; converts self-relative pointers in ActionBlock, MEMNAM, and REGNAM to regular vectors, and Junta's to levStreams to the Init0A procedure.

Init0A()

Initializes the stack, sysZone, and storage area (Storage and EndStorage statics); calls CreateKeyboardStream; then does a GotoLabel to InitRes in Midas.Bcpl to continue initialization.

Init0B(CFA,Switch)

Called from MInit1.Bcpl. Allocates big blocks of storage not saved on the state file. Looks up file FP's needed by Midas using LookUpEntries, forcing /I initialization if any of these don't exist. Calls InitHardware1 to do machine-dependent initialization (should load special Alto microcode, if any, using LoadRAM package and allocate any big blocks of storage not wanted on the state file). Does all initialization reasonable before testing for /I, including building bit buffers and DCB's and allocating storage items that don't change after initialization. On Midas/I does full init (calls OverlayScan, creates any Midas files that don't exist on the disk (Midas.Symtab or Swatee, Midas.Fixups, Midas.Compare, etc.) reads Midas.Programs and Midas.UserPrograms and builds table of FP's and menu for RunProg and RdCmds actions) and returns; otherwise, exits using RestoreState. The Init0 code is flushed after execution.

BlockStoreFP	FP for SWATEE (Alphabetically sorted symbol table)
FixUpFP	FP for Midas.Fixups (temporary file during loading)
ErrsFP	FP for Midas.Compare (output of Compare action)
RunProgFP	FP for Midas.RunProg state file
DtachFP	FP for Midas.Dtach state file

CopyTemplate()

Bogus routine needed by levStreams.

@MBlock

zrel static filled in with pointer to OS MoveBlock procedure.

Storage

Points at the smallest address in the storage area. Initialization code space is added to the storage area by writing the address of the code being flushed into this static.

EndStorage

Points at the largest address in the storage area; decreased as storage is allocated.

StateBlock

Points at the storage block in which the location and value of page-zero statics and some other information is written; the first few words in the block are described by the "State" structure in MDecl.D.

StatePtr
: Pointer into StateBlock.

StateEnd
: ?

StateBlockSize
: contains an upper-bound on 2 * the number of page-zero statics saved in the state file.

FirstStatic,LastStatic,NStatics
: Addresses of first and last statics and number of statics from layout vector.

StateStream
: disk stream for reading state file

RelocTable
: Points at initialization procedure table left by BLDR; used to fixup initialization procedure statics to CallSwat after code thrown away (The fixup sequence in MInit1.Bcpl is presently commented out because the storage for the fixups was overwritten during the initialization.).

* @ACTS
: 100-word vector used in different ways by action overlays.

ProgramStream
: Stream for the left part of the status line.

TimeStream
: Stream for the right part of the status line (only written by PrintTime in Midas.Bcpl).

MStatus
: Vector which holds information saved across RunProg and Dtach actions.

FirstNonOvLine,NNonOvBitBuffers,FirstLCB
: Other items initialized for MDisp.Bcpl.

CmdCS0Vec
: Pointer to the TextVec for CmdCommentStream.

CmdCS1Vec
: Pointer to the TextVec for CmdCS1.

MINIT1.BCPL (second initialization--only called on Midas/I)

Init1()
: Continues initialization begun by Init0. Calls Init0B which returns only on Midas/I. Then, initializes all display regions, symbol table, and EveryTimeList. Enters the registers and memories in the symbol table. Calls InitMPInterface() to build the name-value menus.

NewRegion(blocksize,offset,rlx,rcx,H,W,Type)

returns a new region, with the basic region parameters initialized. blocksize is for the total region including the basic region described as the Rgn structure in MDECL.D. Space is obtained using GetStorage. Arguments are:

blocksize	size of region-description block
offset	nwords from 1st word of block to Rgn structure
rlx	line number of 1st line in region (0 is the null region at the top of the display; 1 to ScreenHeight are the display lines.)
rcx	left-most character in region (0 to ScreenWidth-1)
H	nlines in region
W	ncharacters wide
Type	type of Rgn (see MDecl.D)

MakeMenuRegion(Size,Letter,LineN,rlx,rcx,H,W,MaxNItems)

creates a menu region of the form given by the MDFS structure in MDecl.D using NewRegion to build the Rgn sub-part of the MDFS structure, where:

Size	Size of the area to be allocated
Letter	letter part of menu name for comfiles
LineN	line number part of menu name for comfiles
rlx, rcx, H,W	parameters as for NewRegion
MaxNItems	maximum num menu items

MakeDisplayLine(lvStream,rlx,H,W,rcx)

Creates a text Rgn of height H lines and width W characters beginning at line rlx and character rcx; points Stream at the stream structure for the region; returns a pointer to the TextVec for Stream.

* NewMPDispBlock(L,C,Cnm,W)

must be called to initialize each register display block (see InitMPInterface later)

* GetZStorage(Size)

Allocates a vector of length Size using GetStorage and initializes the body of the vector to 0.

* GetHStorage(Size)

Allocates a vector of length Size+1 using GetStorage and initializes !0 to Size.

MakeTVS(lvStream,lvVector)

Allocates storage for a TextVec of length ScreenWidth and a stream structure using GetStorage; then creates a TV stream. The stream is left in rv lvStream and the vector in rv lvVector.

MakeEmptyTVstream()

Allocates a TV stream without assigning any TextVec to it. Returns a pointer to the stream.

SaveStatics(lvZStatic1,...,lvZStaticN)

Called with up to 20 args. The args are pointers to page zero statics or other program locations which must be saved/restored by SaveState/RestoreState. Non-page-zero statics do

not have to be enumerated because they are all saved.

MINIT2.BCPL (Init2 overlay, also invoked by RunProg)

Init2()

Does final display and region initialization, creates the actions for "RunProg" and "RdCmds" menus; on Dtach, calls the machine-dependent DetachHardware procedure to disconnect from the current machine and closes TextCmdOutStream, if it is open; if not Dtach, parses Com.Cm or the input text line in preparation for RdCmds. Calls the machine-dependent InitHardware procedure, which may return an initialization alternate command menu (e.g., to select among several possible machines); puts non-MPD display lines in service; forms the alternate command menu returned by InitHardware, if any, and then the regular command menu; sets Initialized to true; loads as many overlays as possible into unused core; restores CFileStream, CFOutStream, TextCmdOutStream, and ShowActionForm from the MStatus vector. Returns true if RdCmds should be done.

MTXTBUF.BCPL (resident)

ClearInText()

flags input text buffer to go to empty if any new chars are typed; leaves the static TxtBufClearFlag true.

TxtBNewChar(Char)

puts Char into the command line input buffer (This is usually called via the static CharInputRoutine, which also points at TxtBNewChar.)

* InputTextBuffer

a TextVec containing typed in text

* InputStream

Stream for putting characters into InputTextBuffer

MASM.ASM (resident hand-coded procedures)

DummyCall1(lv Proc,NA,A1,A2,...,An)

calls Proc with NA args so that the overlay machinery can intercept the call and return.

DummyCall(lv Proc,A1,A2, ... , An)

calls Proc with n arguments, A1 ... An, so that the overlay machinery can intercept the call and return.

ErrorProtect(lv Proc, A1, A2, ..., An)

establishes an errorset on the stack to which subsequent calls of DisplayError or ErrorExit will return as discussed in the "Errors and Overlays" section earlier; then calls Proc with n arguments, A1, ... An. If OverlayZone is non-existent or empty when ErrorProtect is called, then overlays are flushed before ErrorProtect returns.

* MoveUp(To,From,Count)
    moves the block of words of length Count with last location at From to the block whose last location is at To (i.e. like MBlock but starting at the last location of the block).

* SelfRel(X) = X + rv X.

* Mag(X) is the magnitude of X.
* LCycle(X,n) left cycles X by n bits.

* VUsc(V1,V2,Length)
    Does an unsigned compare of the DVec's V1 and V2, which are Length words long. Returns 1 if V1 > V2, -1 if V1 < V2, or 0 if V1 eq V2.

* DoubleNeg(Vec) negates the double-precision vector Vec.

DoubleDiv(Vec,Divs) does unsigned divide of double-precision Vec by single-precision divisor Divs; leaves quotient in Vec; returns remainder as the result.

* Wss(S,str) Outputs the string str onto the stream S.
RepPuts(Stream,Char,Count) Does Puts(Stream,Char) Count times

* @WssCSS(str) Wss(CmdCommentStream,str)
* PutsCSS(C) Puts(CmdCommentStream,C)
* ResetsCSS() Resets(CmdCommentStream)

* WssCS1(str) Wss(CmdCS1,str)
* PutsCS1(C) Puts(CmdCS1,C)
* ResetsCS1() Resets(CmdCS1)

ResetTVs(S,TV,N,NoteP,Arg)
    is called by Resets(S,...) when S is a TV stream. Possible calls are:

        Resets(S)
            does TV!0 = 0 using previously-declared TV for the stream.
        Resets(S,TV,N)
            binds the stream to TV of length N and TV!0 = 0.
        Resets(S,TV,N,NoteP,Arg)
            binds the stream to TV of length N and arranges to call NoteP(Arg) whenever the stream is written.

PutTVs(S,Char)
    is called by Puts for TV streams created during initialization. PutTVs optionally executes another procedure, which is MarkRgnDispDirty for some of the streams (see ResetTVs in MTV.BCPL).

PaintItem(rlx,TV,charX)
    is called by MarkMenus to paint a menu item into ScreenTV. TV is a TextVec containing the menu item, rlx and charX are the relative line number and character position of the first character of the item in the region. The parameters of the region were previously established

by a call to PaintBase.

VertIntCode() Vertical interrupt routine initialized in MDisp.Bcpl.

ClearAndScanConvert(BitBuffer,String)
used by MDISP to fill BitBuffer from font characters for String.

StrSize(StrPtr) = length of the string in words.

SearchBlock(Block,Boffset,Bname,Addr,MemX)
is a subroutine which searches Block for the address in MemX whose value is the largest less than or equal to Addr (see MSYM).

SymbKeyComp(Str1,Str2)
compares the two strings returning 0 on equal, positive on Str1 >= Str2, negative on Str1 < Str2.

BinScan(Block,Key)
returns the index of the greatest record in Block with key le Key.

GetBodySize Another subroutine for MSYM.

* GetField(Bit1,NBits,DVec)
Returns the contents of the named field right-justified.

* PutField(Bit1,NBits,DVec,Field)
Copies NBits from Field (right-justified word) to a position in DVec, starting with Bit number Bit1, where the bits are numbered starting from 0. NBits must be le 16.

* MoveLongField(Source,SBit1,NBits,Dest,DBit1)
Moves NBits from the Source DVec to the Dest DVec where the left-most bit in Source is SBit1 and the left-most bit in Dest is DBit1.

PutTxts,ResetTxts
Implement the Puts and Resets stream procedures for display lines (CmdCommentStream, CmdCS1, ProgramStream, and TimeStream). These procedures are invoked by Puts and Resets rather than by direct calls.

BackUp(Stream)

Wait(N)
Idles for 100*N microseconds (N must be > 0)

* @OddParity(X,Y)
Returns true if the number of 1’s in (X xor Y) is odd, else false.

* GetStorage(Size)
returns pointer to permanent block of size Size, or calls Swat if not enough space.

* GetEvenStorage(Size)
returns pointer to block of size Size which starts on an even address.

MaskT
Points 1 before the OS table used by Convert; table entry MaskT!n contains 2^n-1 (i.e., right-justified mask of n bits).

@AbortLabel, @AbortFrame
Statics used by ErrorProtect and DisplayError

@BlockTable
Points at table of pointers to symbol Blocks.

@MSave2,@Count
ZRel statics available for temporary storage by hand-coded procedures.

MDATA.ASM (resident data inappropriate for allocation by GetStorage)

MidasCFA
The CFA for Midas.Run passed to Midas via CallSubsys.

TimeStart is a double-precision vector written with Timer() by StartTimer()

* GoVec is an 8-word vector used by MGo.Bcpl

LastTimer is a double-precision vector used by ElapsedTime(..) in Midas.Bcpl.

TimeTimeOut is a double-precision vector used by the time-out stuff.

BadAText is the default error string for ErrorExit()

MDATAtab
the vector holding the contents of the MDATA memory; this includes a number of vectors of size TValSize used by the "Test" stuff (and may include some machine-dependent stuff too).

@BitsChecked,@ShouldBe,@DataWas,BitsPicked,BitsDropped
Vectors in MDATAtab corresponding to displayable addresses in the MDATA memory.

MADDRtab
the vector holding the contents of the MADDR memory; this includes a number of vectors of length 2 used by the "Test" stuff and may include error counters and other machine-dependent stuff as well).

@LowAddress,@HighAddress,@LoopCount,@CurrentAddress,@AddrIncrement
DWatch,AddrIntersect,AddrUnion,TestFailures
Vectors in MADDRtab with corresponding displayable addresses in the MADDR memory.

MSYM.BCPL (resident--symbol table and parsing)

EvalAText(TV,lvX,AVal,ifExpectMore,SName)

will return true if the text starting at TV!(rv lvX) has the form of a memory name (optional Offset), register name, or address (optional +/- Offset), and, unless ifExpectMore, is not followed by more text. SName (optional, defaulted to DefMemName) is the default memory name, used if the first term in TV is a pure number rather than a symbol. The optional Offset may be any expression consisting of integers and address symbols connected by "+" or "-"; this expression is evaluated using the default radix for the memory (obtained from MEMCON).

In the case of a true return, AVal (see AVal structure in MDecl.D) is filled with information describing the address; rv lvX is adjusted to point to the next field, after being stepped over one character (assumed to be an appropriate separator).

SkipBlankToken(TV,lvX)

advances rv lvX past leading blanks starting at TV!(rv lvX).

ChkToken(TV,lvX,lvSize)

Begins parsing at the (rv lvX)th item in TV. It advances rv lvX past leading blanks, parses the token following that, leaving the size in rv lvSize, and returns the kind of token as its result. The tokens returned are given in a manifest in MDecl.D. These are arranged so that numbers of varying radices can easily be distinguished.

StreamFromTextName(TV,DotExt,ksType,ItemSize)

Parses a file name in TV, defaulting the extension to the string DotExt; then calls QuickOpenFile. Returns the stream on success, else calls ErrorExit()

QuickOpenFile(Name,ksType,Item)

searches the table of file DV's built from Midas.Programs and Midas.UserPrograms and opens a stream using CreateDiskStream, if Name is found in the table or using OpenFile, if it is not.

TVtoString(TV)

returns a string with same contents as the textvec TV. Only one such string may exist at a time. (They are stored in a fixed vector @StringVec) useful for making entries in symbol table.

* SearchBlocks(S,MemX,AVec,IMA,MemNameP,Radix)

searches all symbol blocks for the address symbol in MemX whose value is the largest le AVec (currently limited to MemX's that have single-precision AVec's). Optional IMA arg causes ".-3" to ".+3" to be printed if no symbol exactly equals AVec and if AVec!1 is within 3 of IMA. Otherwise, the symbol is printed on the stream S followed by the displacement "+n". If the optional arg MemNameP is omitted or true, AVec is printed as "memname val" when no address symbols are in the table, else just as "val". Radix controls numerical printout.

Before searching all the symbol blocks, the machine-dependent procedure FastSearch(Addr,MemX) is called; FastSearch may return the BlockTable index of a block

known to contain the best symbol, thereby reducing compute time.

MapSymBlocks(Proc,Strategy,A1,A2,A3,A4)
: Mapping procedure to apply Proc(B,A1,A2,A3,A4) to each of the symbol blocks, where B is a pointer to the BT structure for the symbol block (in BlockTable). When blocks need to be read from the disk, the Strategy argument is used in the call to GetBlock. MapSymBlocks is primarily for use by SearchBlocks, but machine-dependent code may use it to build inverted tables for FastSearch.

FindInTable(Key,Body,lv BodySize)
: attempts to find a table entry. Returns true if succeeded. Key is a pointer to a bcpl string. Body is place where result record will be placed. BodySize will contain actual size of body.

GetBlock(BlockAddr,Kind,Strategy,Block)
: Ensures that the record file block with index BlockAddr is in core, swapping from the disk if necessary. Returns a pointer to the BT descriptor for the block. If BlockAddr is not in core, FindFreeBlock(Block,Strategy) is called to pick an in-core block for replacement; the block selected will have a core address different from Block (which should be 0 if don't care).

FindFreeBlock(Block,Strategy)
: returns the BlockTable index of a free core block different from the one at core address Block using Strategy to select among the contending blocks; calls PutBlock to write the selected block, if it is dirty. Strategy is a small integer selecting the replacement strategy as discussed in the listing.

Map
: points at a table of length MaxBlockPages+1 whose entries contain the disk addresses for pages in the record file (or FillInDA); used by GetBlock and PutBlock.

@BHsize contains the size of the BH structure in MDECL.D

@StringVec points at a 129-word vector used by TVtoString.

MSYMOV.BCPL (in Loader overlay)

UpdateRcd(Key,Body,BodySize,nil)
: either inserts a new or updates an existing symbol table record.

PutBlock(B)
: Writes the block whose BT descriptor is at location B onto the disk.

CleanUpBlocks()
: Writes all dirty record file blocks ont the disk.

MIOC.BCPL (resident I/O conversion to/from streams/DataVec's)

* DWns(S,DVec,NBits,DBit1,Radix,Width,flush0)
    Outputs the number contained in a field of a DVec onto a stream S. NBits is the number of bits in the field (default 32); DBit1 the first bit (default 0); Radix is defaulted to 8 (negative radix means print as a signed number); Width (default 1) is the minimum number of characters printed; flush0 controls the fill characters output in front of the number and the handling of a number with value 0: flush0 eq 0 means don't print anything if the number is 0; flush0 eq $*S means supply leading blanks to fill Width; flush0 eq $0 means print leading 0's to fill Width. DWns returns flush0 as its result if the number is 0, else it returns $0.

* Wns(S,num,width,radix)
    Same as OS but uses DWns (The OS procedure has been Junta'd away).

* SimpleTexttoDVec(TV,Nbits,DVec,Radix)
    converts a TextVec into a DataVec of size Nbits. Blanks are flushed. If characters other than octal chars and blanks are in TV, then the text is treated as an an address and offset which is evaluated. Radix is defaulted to 8.

GenlTexttoDVec(TV,Form,DVec,Radix)
    parses TV into a sequence of expressions delimited by blanks; matches these expressions right-to-left with corresponding fields in the format vector Form (from MEMFORMS or REGFORMS in xxTables.Asm), evaluates the expressions, and stores the values into the fields of DVec.

* DatatoStream(Stream,Form,Width,DVec,Radix)
    format data in DVec and output the text to Stream under control of Form (from MEMFORMS!MemX or REGFORMS!RegX) or use Width (from MEMWID!MemX or REGWID!RegX) if Form is 0. Radix is defaulted to 8.

MOVERLAY.BCPL (resident overlay manager)

@OverlayFaultProc(ac0,ac1, . . .)
    intercepts calls to procedures not in core and swaps in the overlay containing the procedure.

ReadOverlay(pn,core,np,od)
    reads an overlay; used by OverlayFaultProc and by Init2 (see code for details).

KillOverlays()
    will cause all overlays in OverlayZone to be flushed and ReUseDispSpace() to be called (no-op if OverlayZone contains 0).

FlushOverlays()
    will cause all overlays in OverlayZone to be flushed, but leaves OverlayZone intact.

PeelOverlay()
    Called to remove the top-most overlay to make space for a new symbol block or in preparation for RunProg.

SwappedOut()
: Dummy routine needed by BLDR; after initialization, procedure statics in initialization code are bound to this procedure.

OverlayZone
: Zone from which overlay space was obtained, can be used for space by the overlay code, so long as all use terminated when overlay flushed**

OverlayFlushed
: set true by KillOverlays, false by ReUseDispSpace. Indicates that the bit buffers in AvailBlock have to be rebuilt and OverlayZone destroyed.

OvTable
: table of overlay od's manipulated by overlay loading code in MINIT2 and by PeelOverlay().

MDISP.BCPL (resident display driver)

UpdateDisplay(Z)
: The Z arg is passed only for the call from SetDisplay(false); unless this arg is passed, UpdateDisplay will return without doing anything when the display is off. If the display is on or if the Z arg is passed, then PaintDirtyRegions is called to rebuild ScreenTV for text lines. Then GetFreeBitBuffer is called to rebuild bit buffers for all the display lines.

PrepareCharInv(charlocinfont)
: is called by ClearAndScanConvert in MAsm.Asm to invert black for white in a font character.

FinishDisplay()
: is called from MidasFinish() before returning to the Executive.

* SetDisplay(Flag,nil,nil)
: called both directly and as an action; turns off the display if Flag is true, turns it on if Flag is false. Result is previous value of DisplayOff. If the display is being turned on and OverlayFlushed is true, ReUseDispSpace() is called; then UpdateDisplay is called.

* Blink()
: will blink the entire display once for about 0.2 sec.

MakeDispZoneAvail()
: blanks the part of the screen used for register name-value display, and creates OverlayZone from the bit buffers of that portion of the screen.

ReUseDispSpace()
: reverses the effect of MakeDispZoneAvail(). Used only by KillOverlays() in MOverlay.Bcpl and by SetDisplay(false).

VertIntFlag (referenced in MAsm.Asm) indicates that a vertical interrupt has occurred since the flag was set false--used by the bit-buffer management stuff to determine when an old bit buffer

can be released.

NwdsPerScanLine
: the number of bit-buffer words/scan line

FontCharWidth,LineCtrlBlockPtrsVector

DisplayOff
: A predicate that is true when the display is off; it is manipulated by SetDisplay(..)

GACHA10.BR (resident GACHA10.AL font built by the MakeFont program)

FontP
: Points at the second word of the font. FontP!-1 contains the font width (fixed pitch) in the right-half. FontP!-2 contains the height of the font in scan lines.

MRGN.BCPL (resident display region code and main control program)

DriverLoop()

This is the main loop of Midas also called by StartCmdOverlay. The top level returns only for exit from midas. QuitCmdOverlay is used to exit from actions running under StartCmdOverlay. DriverLoop calls UpdateDisplay when screen lines are dirty, CharInputRoutine(Gets(keys)) for keyboard input, tracks mouse movement, and for executing various routines when the mouse moves to different display regions or has its buttons clicked.

PaintSetup(Rgn,rlx)

Sets up the DisplayMaxrcx, Displayalx, and Displayacx statics to delimit the display area covered by the line rlx in the region Rgn and marks the screen line dirty; returns the screen line as its value.

PaintMark(R,rlx,firstC,lastC,Flag)

clears or sets the black-for-white invert flag in the characters for a particular menu item. R is the display region, rlx the relative line number, firstC and lastC the relative character positions in the line, and Flag is 200B for invert, or 0 for non-invert.

PaintRgnLine(R,rlx,TV)

moves text from TV into ScreenTV for region R (called by PaintDirtyRegions and by FormCmdmenuText).

PaintDirtyRegions()

Moves text from TextVec's to ScreenTV for all regions which are not menu regions (Menu regions paint directly into ScreenTV.)

UpdateEveryTime()

calls the procedures on the EveryTimeList.

* AddToEveryTimeList(Proc,Arg)

assures that a call will be made to "Proc(Arg)" once every time around the main driver loop. AddTo... returns as result a pointer which can be used as an argument to RemoveFromEveryTimeList.

* RemoveFromEveryTimeList(Entry)

will stop the calls resulting from an earlier call on AddToEveryTimeList, where that earlier call returned Entry.

@NewLx Cursor line index

@NewCx Cursor character index

Displayalx, Displayacx, DisplayMaxrcx, CharInputRoutine

ErrorFlag

A predicate set true while DisplayError is in progress. When true, it prevents calls to EveryTimeList procedures and to Action procedures outside the command menu region

(which is showing the error menu).

AllowedRgn
: The allowed region (command menu region) during an error.

ScreenLinesDirty
: Set true when UpdateDisplay should be called.

ScreenTV
: A vector of length ScreenHeight+1 in which ScreenTV!0 contains ScreenHeight and the other entries point at strings of length ScreenWidth. The bit buffers are built from these strings.

SelectedRegion
: The currently selected display region.

RegionTable
: Table of pointers to the Rgn structures for each region. The first entry is NopRgn (the entire active display area). Then come the name-value menu regions, which must be contiguous pointers in the table. The other regions come after this.

MMENU.BCPL (resident menu management)

* CreateAction(Name,lv Proc,Arg,lv MProc,Char)
: creates an action for use in a menu. Name is a text name for the action. Proc is a procedure. (lv is used so if the contents of the static changes, e.g. during swapout, the action will use the changed value of the static.) Arg is supplied as Arg1 in the call to Proc. The five-word Action structure is appended to ActionBlock and the name string is not copied, so it must be resident during the lifetime of the action. If lv MProc is given and non-zero, MProc will be called each time mouse buttons or position change and this action is the selected menu item. Char is a character which invokes the action from the keyboard when ";" precedes it (and "=" and ":" not preceded by ";")

ForgetTemporaryActions()
: Deletes actions created during command overlay (normally called from QuitCmdOverlay() in MCmd.Bcpl).

DoAction(Action,MBunion,MDFS)
: Executes Action. Action points at a 5-word ActionBlock entry which contains the procedure called (Action.lvProc) and some flags that control whether or not the command comment lines are reset and the handling of a subsequent <esc>. MBunion is the union of all buttons down since last up; MDFS is the menu in which the action is executed.

DoTextAction(Char)
: exectutes the action whose command character is Char in the command menu.

ExecuteTextCmdStream(Stream)
: is used to read commands from a text stream, of the form created by the WrtCmds.

LookUpMenu(Letter,LineN)

Letter is the menu letter ($A to $Z) and LineN is the number of the first line in the menu or -1 if no line number is associated with the menu. Returns 0 if the menu doesn't exist, else returns a pointer to the MDFS structure for the menu.

FormMenu(MDFS,Proc,nil)

rebuilds a menu by initializing and then calling Proc(ItemStream,MDFS). Proc builds each item in the menu by writing text on ItemStream and calling MarkMenus. Usually, composite procedures discussed below are used to do this.

MarkMenus(Action)

is called to mark the last text output to ItemStream as a menu item such that Action gets called when that text is bugged. Marks as a null menu item if Action is 0. Advances to the next line in the menu region if no text has been output to ItemStream. Finishes by reseting ItemStream.

* @WsMarkA(Action)

appends the name of Action to ItemStream and calls MarkMenus; then marks a single blank as a null Action.

* WssMAct(Action)

appends the name of Action to ItemStream and marks Action for execution.

* WssMark(Name,Action)

appends the string Name to ItemStream and marks action for execution.

MenuMChange(NewSelRgn,InRgn,NewMB,MBunion)

is called by DriverLoop() in MRgn.Bcpl when the currently selected region is a menu region and the mouse buttons or character position changes.

deSelectMenu(R)

is called by DriverLoop() when the mouse moves out of the selected region and the selected region is a menu region.

MMPRGN.BCPL (resident name-value menu management and actions)

* UpdateMPDValues(AlwaysOnly)

must be called after register values change. UpdateMPDValues() loops through all of the MPD menu areas on the display and for each one that isn't empty calls MGetRegData or MGetMemData to obtain the current value. If the new value differs, it sets a flag so that the text in that menu area will be rebuilt later by MPDEveryTime(..). The AlwaysOnly argument is normally omitted; when it is true, only those MPD items that have AlwaysUpdate true in MEMCON/REGCON are updated. This alternate use of UpdateMPDValues aims at machine state that should be polled and updated continuously on the display, even when the machine is running or is at a breakpoint (Most hardware implementations will not have any registers or memories marked with AlwaysUpdate).

MPDdeSelect(MPD)

deselects an MPD menu.

MPDMChange(lvTable,MPD,MBunion,MB)
: does the menu switching for the name-value menu MPD in response to the mouse buttons going up and down.

* BadAltIn(TV,DVec,BadStr)
: Used as a filler for AltMInput and AltRInput table entries when an alternate input procedure has not been defined.

* RDatatoCSS(RegX,DVec,AVec,ExtRadix)
: = DataToStream(CmdCommentStream,form,REGWID!RegX,DVec,Radix)
This is the default pretty-print routine in the AltRInput table, used on registers for which no special print routine is implemented.

* MDatatoCSS(RegX,DVec,AVec,ExtRadix)
: =DataToStream(CmdCommentStream,form,REGWID!RegX,DVec,Radix)
This is the default pretty-print routine in the AltMInput table, used on memories for which no special print routine is implemented.

ShowAddr(AVal)
: pretty-prints any address item in a name-value menu in standard form and then calls HWShowAddr(X,AVec) to do any additional pretty-printing.

GetRadix(MPD)
: returns the radix for the item in the name-value menu MPD.

MMPRGNOV.BCPL (Command overlay)

Implements less common name-value menu actions.

MCMD.BCPL (resident command menu, error handling, etc.)

* WnsCSS(N) = Wns(CmdCommentStream,N,0,8)
* WnsCSSD(N) = Wns(CmdCommentStream,N,0,10)
* WnsCS1(N) = Wns(CmdCS1,N,0,8)
* WnsCS1D(N) = Wns(CmdCS1,N,0,10)

* ConfirmAct(Str,TV)
: temporarily turns on display (if off) and prints on the command comment line first Str, then TV, and finally " [Confirm]". Returns false if not confirmed else true if confirmed, display restored to original state.

* DisplayError(S1,S2,S3,NoCSS)
: is called by action procedures to show an error message on the display and wait for the user to abort or continue. If called during Midas initialization, it does a CallSwat(S1). Otherwise, unless the fourth arg NoCSS is passed it does ResetsCSS(). Then it shows S3 (if given)

followed by S1 on CmdCommentStream and calls BeginError. S3 and S1 explain the error. S2 is an optional string which will be a selectable item to allow continuation. The only other selectable item aborts.

* ErrorExit(S1,S2,S3)

does CallSwat(S1) if it occurs during Midas initialization. Otherwise, it shows S1, S2, and S3 on CmdCommentStream, if given. Then, if the call occurs during a command file or when the display is off it calls DisplayError(0,0,0,true). Otherwise, it aborts the command overlay (if any) or returns to the last ErrorProtect call (if no command overlay).

* ErrorAbort(S1,S2,S3)

similar to ErrorExit, called by procedures that are not in command overlays; exits to the last ErrorProtect call; a procedure distinct from ErrorExit is needed so that actions erroneously executed during a command overlay (such as SetValue) will be aborted without aborting the command overlay.

* SetAbort(lv Proc,Arg)

binds the "Abort" action (alternatively, control-C) to Proc, where Proc is frequently CmdAbort.

* SetAbortPure(lvProc,Arg)

same as SetAbort except that CmdCommentStream and CmdCS1 are not reset at the onset of the action.

CmdAbort(nil,nil,nil)

procedure frequently bound to the "Abort" action. It does ResetsCSS(), prints "XXX" on CmdCommentStream, and then aborts the command overlay or (if not command overlay) the last ErrorProtect call.

* FormCmdMenu()

Rebuilds the command menu.

ShowActions(Act1,FirstAct,NActs)

calls WsMarkA(Act) for Act = Act1 and for the NActs actions starting at FirstAct in the ActionBlock.

* StartLargeOverlay(lvProc,A1,A2,A3,A4)

is used to execute an action in a very large command overlay. Proc is called first to build the actions; Proc returns lv XProc as its result, where XProc is in a different overlay; OverlayZone is cleared after Proc returns. Then a large amount of extra storage from the stack and sysZone is added to OverlayZone and StartCmdOverlay(lv XProc) is carried out; overlays are flushed, extra storage from sysZone released, and OverlayZone rebuilt after StartCmdOverlay returns.

* StartCmdOverlay(lvProc,A1,A2,A3,A4)

is used to call a procedure which returns a procedure for forming an alternate command menu. The alternate menu is used until some action in that menu calls QuitCmdOverlay.

* QuitCmdOverlay(Val)

is called by some action in an alternate command menu to exit from the original call to StartCmdOverlay and resume the normal command menu; Val is returned to the caller of StartCmdOverlay (DisplayError and ErrorExit return 0 to the caller of StartCmdOverlay.).

RestoreState(FP,reinitFlag)

restores "state" from the file with file pointer FP, which should correspond to either Midas.RunProg or Midas.Dtach written by CmdWriteState. reinitFlag is true when RestoreState is called as part of program initialization, false when call is to reinitialize for RunProg. It finishes with GotoLabel(TopFrame,Resume,0) which sends control to the Resume label in the InitRes procedure in Midas.Bcpl.

* CmdCommentStream

A stream for writing comments to the first comment line; ResetsCSS, PutsCSS, WssCSS, WnsCSS, and WnsCSSD are procedures that execute standard stream operations on CmdCommentStream.

* CmdCS1

A stream for writing comments to the second comment line; ResetsCS1, PutsCS1, WssCS1, WnsCS1, and WnsCS1D are procedures that execute standard stream operations on CmdCS1.

TextCmdOutStream

if non-zero, a stream on which DoAction should write logical actions.

ShowActionForm

if true, then the command file form of each action will be shown on the second comment line each time a new action is selected.

SavedLoadText

Vector of ScreenWidth words retaining file names of last load for use by subsequent Dump

LoadDone

True to enable the "Dump" action (after a previous load)

ProgramStream

One text-line stream between the register display and the command comment line on which the name of the last program loaded is displayed. Initially, the release dates of the machine-independent and machine-dependent parts of Midas are displayed here.

CmdAltMenuP

Non-zero if an alternate command menu is up (needed by TxtBNewChar to check for illegal type-ahead).

MCMDOV.BCPL (Command overlay)

Implements action procedures referenced only in xxActions.Asm. These actions are only executed during command files.

MBRKP.BCPL (Brkp overlay)

InsertBreak(MBunion,AVec,MemX)
: is an action procedure called from StartWithAddr to insert a breakpoint (The machine-dependent BreakIML procedure does the work).

RemoveBreak(MBunion,AVec,MemX)
: like InsertBreak but removes a breakpoint.

MGO.BCPL (Resident--SS, Go actions)

* CallWithAddr(lv Proc,MBunion,nil)
: is an action routine used to collect an address from the input text line and then do either DummyCall(lv Proc,MBunion), if the input line is clear, or DummyCall(lv Proc,MBunion,AVec,MemX), if the input line evaluates to an address in MemX. Used to call SingleStepM (SS action), InsertBreak (Brk action), and RemoveBreak (UnBrk action).

* StartWithAddr(lv Proc,MBunion,nil)
: like CallWithAddr, but uses StartCmdOverlay instead of DummyCall. Used to call StartM ("Go" action), RepeatGo ("RepGo" action), RepeatSS ("RepSS" action), and OpcodeStep ("OS" action) for Dorado.

* StartBigOvlWithAddr
: like CallWithAddr, but uses StartLargeOverlay instead of DummyCall. Used to call StartSim (SimGo action for Dorado).

* HaltWait(nil)
: The EveryTimeList procedure which waits for the machine to halt and calls MStopped(true) when it does; meanwhile updates the compute time on the display. Used by the indefinitely-computing Go actions.

* HaltWaitMenu(nil,nil)
: Menu-forming procedure that shows an alternate command menu consisting of the single action "Abort."

DetachMenu(nil,nil)
: Menu-forming procedure that shows both "Abort" and "Dtach."

* StartSetup(EveryTimeP,EveryTimeA)
: binds the "Abort" action to the HaltProc procedure, adds EveryTimeP(EveryTimeA) to the Every-Time List, and returns DetachMenu as its value.

* HaltProc(nil,nil,nil)

used as the "Abort" action for all the indefinitely-computing "Go" actions (includes "Call", "OS", and "SimGo" on Dorado).

@CantContinue

The static used as a one-word bit table in which each bit represents a reason why it is impossible to continue execution of the microprogram from its last breakpoint or keyboard halt. Manifests in MCommon.D include both machine-independent and machine-dependent definitions for the bits. The MStopped procedure in xxGo.Bcpl first clears CantContinue and then may set some bits, if reasons for not continuing are detected; other procedures "Or" reasons into CantContinue. The SetupIMA procedure in xxGo.Bcpl will print the various reasons and wait for confirmation if the user tries to continue.

MGOOVL.BCPL (GoOvl overlay--Call)

* StartWithArgs(lvProc,MBunion,nil)

is an action procedure used to collect the starting address and arguments for the "Call" action. If FOO(a1,a2, ... , an) are on the input text line, FOO is evaluated to a memory address stored in CallAVec!0 and CallAVec!1 and the MemX for FOO in CallAVec!2. a1 to an are evaluated to 32-bit integers stored in a vector. Then StartCmdOverlay(lvProc,MBunion,AV,n) is called where AV contains the arguments.

MRGO.BCPL (RepGo overlay)

RepeatGo,RepeatSS actions

MLOAD.BCPL (Loader overlay--load, compare, dump)

InitLoad(lvProc)

brings in loader overlay and invokes lvProc after initialization, where lvProc points at one of the following:

LoadMB	(Ld action) Loads .MB files specified by input text
LoadSyms	(LdSyms action) Loads symbols only
LoadData	(LdData action) Loads data only
DumpMB	(Dump action) Dumps microprocessor state on .MB file specified by input text under control of .MB files used for previous LoadMB, LoadSyms, or LoadData.
CompareMB	Compares microprocessor state against .MB file where .MB file must not have any fixups.

MPATTERN.ASM (Resident--in Debug overlay except on Dorado)

* CheckData()
  Increment double-precision LoopCount; then compares the DataVec's of size TValSize in ShouldBe and DataWas. If bits selected by BitsChecked are unequal, then accumulate failure information in AddrIntersect, AddrUnion, BitsPicked, BitsDropped, and TestFailures (discussed in user manual), and return -1; otherwise, return 0.

* NextData()
  Generates the next data pattern of size TValSize (as selected by SetPattern) and stores it in the DVec ShouldBe.

@RANDIX
  variable used by the random number generator (must not smash it).

@PATTERN
  variable used to control NextData but available for other uses at other times (used extensively by pretty-print procedures).

RanTab
  Vector of length RanLen used in random number generation.

RanLen
  Length of RanTab.

MDEBUG.BCPL (Debug overlay)

* GetPattern(NumActs,DebugProc)
  Creates the data-pattern-select Actions, sets the command abort action to be the TestStop procedure, and returns DebugMenu to StartCmdOverlay. DebugMenu is then called to build the pattern-select menu. After a selection is made, DebugProc is called from DebugMenu to display the test-select menu (or whatever). NumActs is stored in the static NActs, then used by DebugProc.

* ShowPattern(S1,S2,Radix)
  Prints S1, S2, " with ", the pattern name, and ", mask = " on CmdCommentStream; then it prints BitsChecked using the format vector for the item being tested.

* SetupPattern(DMask,Name,Radix)
  The caller must initialize BitsChecked when the new item being tested differs from the last; also, the caller must put a right-justified "1" into IncV. SetupPattern will then mask BitsChecked by DMask and use IncV in controlling the SEQUENTIAL, CYC0, and CYC1 patterns; it initializes LoopCount, AddrUnion, AddrIntersect, TestFailures, BitsDropped, and BitsPicked and sets up ShouldBe for the data pattern. It then resets CmdCommentStream and calls ShowPattern.

* ErrorStop(str1,str2,AVec)
  is used to print a message on CmdCommentStream and exit from a test.

CmdCommentStream is reset, the str1 and str2 are printed, and finally AVec (a double-precision integer) is printed (if given).

WriteComment()
: Resets and then writes the strings pointed to by STAT1 and STAT2 on CmdCommentStream; if STAT3 ne -1, it is also printed onto CmdCommentStream as an octal number

* SaveDGen()
: saves the state of the pattern generator in the Vec60 table.

* RestoreDGen()
: restores the state of the pattern generator from the Vec60 table.

* ContinueGen(lvProc,Arg)
: is used to continue a previously-aborted test. This is called when <esc> has been typed after a test was aborted or halted due to an error.

MDEBUGASM.ASM (Debug overlay)

PatTab
: Table of self-relative pointers to the names of the data patterns that appear in the menu of possible data patterns

NPatterns

MTEST.BCPL (Test overlay)

All of these action procedures are called from StartCmdOverlay. Testing of registers, memories, and other items is controlled by the "TestAll" bit in REGCON/MEMCON tables and by the various tables in xxTestAsm.Asm.

DataTest(nil,nil)
: defines all actions in the "Test" menu and the data-pattern-selection menu; returns the result of GetPattern as the menu-forming procedure.

ContinueTest(nil,nil)
: implements <esc>action after interrupted "Test".

FreeRunTest(nil,nil)
: implements <cr> action after interrupted "Test".

TestAll(nil,nil)
: implements the "TestAll" action.

ContinueTAll(nil,nil)
: implements <esc> after interrupted "Test-All"

SkipTAll(nil,nil)
implements the <cr> action after interrupted "TestAll".

LastVal

MFIELDLP.BCPL (Field overlay)

Implements the "Fields" action

MCOLLECT.BCPL (LDRtest overlay)

* CollectLDR(Vec,MemX)
Collects a list of up to six addresses from the input text line and ensures that they are in MemX; if not ErrorExit is called. This procedure is provided for use by any machine-dependent actions that require lists of addresses (used for "LDRloop" action on Dorado).

MPESCAN.BCPL (PEscan overlay)

PEScan() implements the "PEscan" action
Those memories that contain parity (indicated by bit in MEMCON) appear in a menu that allows the user to select which ones are scanned for parity errors.
ScanForPE(MemX,WaitP,Vec) is called for each memory selected.

* ReportPE(Vec,NPEX,DVec,MemX,AVec,WaitP)
is called by ScanForPE to report parity errors. Vec, MemX, and WaitP are the arguments passed to ScanForPE by PEscan; AVec is the address in the memory (double-precision vector); NPEX is ...

MPRINS.BCPL (resident--procedures for pretty-print overlays)

PrinV0(String,Value)
prints "String=Value" on CmdCommentStream, where Value is a 16-bit integer printed in the default radix (DefRadix).

PrinV1(String,Value)
like PrinV0 using CmdCS1.

NWss(String,BitMask)
prints "String" on CmdCommentStream iff (PATTERN & BitMask) ne 0, where the BitMask argument is defaulted to NextPattern if omitted; then does NextPattern = BitMask rshift 1.

NWss1(String,BitMask)
like NWss using CmdCS1.

xxACTIONS.ASM (resident--table of Actions)

Defines many statics that point to actions

ActionBlock
: points at the first action in the table.

ActionPtr
: the number of actions currently in the table.

LastAction

FirstCmdAction,LastCmdAction
: point at the first and last actions that are in the primary command menu.

NHWCFActions,HWCFActions
: HWCFActions points at the first RdCmds action for the file Special.Midas; NHWCFActions is the number of entries to Special.Midas.

EscAction,NewEscAction,CRAction,NewCRAction
: are manipulated by DoAction(..) in MMenu.Bcpl and by the test stuff (potentially elsewhere) to determine which action is executed in response to the user typing <cr> or <esc>.

@LongOne,LongMinOne.

xxTABLES.ASM (resident--register and memory tables)

Defines register and memory description tables discussed earlier.

xxREG.BCPL (resident--register read-write)

PutRegData(RegX,DVec)
: Returns false if writing register RegX is illegal; otherwise, stores DVec into RegX and returns true; comments similar to PutMemData apply.

MPutRegData(RegX,DVec,nil,Extension)
: If writing is legal, then calls PutRegData(...), restores incidentally clobbered registers, calls UpdateMPDValues, and returns true; otherwise, calls ErrorExit().

GetRegData(RegX,DVec)
: fetches register RegX into DVec and returns true if ok, false if the value cannot be obtained for some reason.

MGetRegData(RegX,DVec,nil,lvExtension)
: Calls GetRegData(...), restores incidentally clobbered registers, and returns true if ok, false if the value cannot be obtained for some reason.

xxMEM.BCPL (resident--memory read-write)

CertifyAV(MemX,AVec)
: Is called from EvalAText to verify that AVec contains a legal address in MemX; returns true if ok else false (in D1ASM.ASM).

PutMemData(MemX,DVec,AVec)
: stores DVec in MemX, AVec. Returns true if operation went ok, else false. PutMemData, which does not restore registers clobbered incidentally, is called directly by Ld and LdData actions and by Test actions, where speed is more important than cleaning up; in most other contexts, MPutMemData is used instead. Even though it doesn't have to restore registers, PutMemData must leave the hardware in operable shape and not clobber any memory words except those considered to be volatile during program execution.

MPutMemData(MemX,DVec,AVec,Extension)
: If AVec is a legal address in MemX and writing is legal, then calls PutMemData(...), restores incidentally clobbered registers, calls UpdateMPDValues(), and returns true; otherwise, aborts with ErrorAbort. Since CertifyAV has verified the legality of AVec, the reason for failure will normally be something such as read-only violation, can't write because machine is running, etc. Extension is 0 and ignored except for items that extend to more than one MPD region on the display (see the ROW or PIPE memories in Dorado Midas for how Extension is used).

GetMemData(MemX,DVec,AVec)
: fetches data from AVec in MemX, stores the result in DVec. Returns true if successful, else false; volatile registers used in the course of the Get are not restored. GetMemData is called directly by Dump and Test actions; in most other contexts MGetMemData, which restores registers clobbered during the Get, is called.

MGetMemData(MemX,DVec,AVec,Extension)
: If AVec is a legal address in MemX and reading is legal, then calls GetMemData(...), restores incidentally clobbered registers, and returns true; otherwise returns false.

xxRES.BCPL (resident--register memory read-write)

FormHWMenu()
: builds conditional parts of the machine-dependent command menu (Unconditional menu items are in the part of ActionBlock managed by FormCmdmenuText).

DetachHardware()
: called to do any machine-dependent stuff needed for the "Dtach" action (called "Boot" on D0 Midas); called by FinishHardware and by Init2.

FinishHardware()
: Routine called before Midas exit to the Executive.

HWShowAddr(MemX,AVec)
: called from ShowAddr in MMPrgn.Bcpl to do special additional pretty-printing for the

"Addr=" action.

HWAlwaysUpdate(MorRCON,X,AVec)
: returns 1 if the memory or register identified should be "always updated" on the display; MorRCON is the contents of the MEMCON/REGCON table entry for the memory/register; this procedure is needed because some of the addresses in the MADDR memory are always-updated while others are not, so cannot simply check the MEMCON/REGCON bit to decide.

xxBRKP.BCPL (BrkP overlay)

BreakIML(AVec,MemX,BPflag,String)
: is called from InsertBreak with BPflag true and from RemoveBreak with BPflag false; AVec eq 0 if the address should be defaulted, else MemX and AVec define the address at which the break point should be inserted or removed; String, then " break at ", then the address are printed on CmdCommentStream.

xxGO.BCPL (resident--procedures called by MGO.BCPL)

SetupIMA(GoP,AVec,MemX,NA,String,MBUnion)
: is called by RepeatGo, RepeatSS, SingleStepM, and StartM. It prints String followed by other information on CmdCommentStream if the address is ok and aborts otherwise. GoP is a predicate meaning go full speed if true, single-step if false. AVec and MemX are the address information for the go or step. NA is 1 if AVec should be defaulted to continue from the last break, else 3. MBUnion distinguishes a "new go" from a "continue"; by convention BottomButton is true when the control section should not be reset before starting; otherwise, io devices and the control section should be reset, so that only the task being started is active. The static GoVec is a vector in the form of the GoVec structure in MCommon.D (which contains both machine-independent and machine-dependent stuff); (GoVec>>Go.Proc)(GoVec) is called before returning; the caller will then start the machine with OneStep(GoVec>>Go.RunP).

MStopped(GoFlag)
: Is called when Go, SS, Call, OS, RepGo, or RepSS terminate. It should setup to read the machine state, and print on the comment lines information about the location of the halt and reason for halting. If the GoFlag argument is true, then the reason for halting is printed. If the GoFlag argument is true or omitted from the call, then exit with QuitCmdOverlay().

DefaultGoMemory()
: Establishes the default memory for a Go, SS, etc. action so that number;G on the input text line will be evaluated to an address in the correct memory.

OneStep(GoArg)
: is called after SetupIMA has done the primary setup for a Go, SS, etc. It either starts the machine running full speed or single-steps it, according to GoArg.

Stop()

Halts the hardware, assumed to be running (in D1ASM.ASM).

@CheckStopped()
Returns true if the hardware is running, else false (in D1ASM.ASM).

xxLOAD.BCPL (Loader overlay)

PrepareLoad(SymOnly)
Verify that the hardware is in good shape for a Ld, LdData, LdSyms, Cmpr, or Dump action and do other setup. Call DisplayError if probably not ok to execute the action; if DisplayError returns (indicating user wants to go ahead with the operation), then call ResetsCSS().

RestoreAfterLoad()
cleans up after Ld, LdData, LdSyms, Cmpr, or Dump actions; checks for errors detected during the Ld and calls ErrorAbort if there were any problems, else returns.

PutMDSymOnly(MemX,DVec,AVec)
kludge routine replacing PutMemData on the LdSyms action. This is needed on the Dorado and D0 implementations of Midas so that the virtual memory table can be loaded; other implementations of Midas will probably define this as a no-op.

AddToVM(VA,DVec)
Discussed below in the xxVM.BCPL section (part of the virtual control store stuff used for Dorado and D0 implementations).

LoadCleanUp()
Called after a load to build inverted symbol tables for use by the FastSearch procedure.

xxDEBUG.BCPL (Debug overlay)

QuitTest()
is called by ErrorStop. It should prepare for subsequent calls to MGetRegData and MGetMemData, call UpdateMPDValues and exit with QuitCmdOverlay.

xxTEST.BCPL (Test overlay)

ValidateTest(X)
Sets up for testing the register, memory, or other item denoted by X, where X = 0 to NREGS-1 denote registers, NREGS to NREGS+NMEMS-1 denote memories, and NREGS+NMEMS to NREGS+NMEMS+NOther-1 denote other tests.

Other procedures implementing the "other-test" items.

xxTESTASM.ASM (Test overlay)

Defines tables for "Test" and "TestAll" actions discussed earlier.

xxFIELD.BCPL (Field overlay)

FieldLoop(FieldX)
: Should make the hardware execute the microinstruction setup by SetupFieldTest(..) and ModifyField(..).

SetupFieldTest(FieldX)
: Should prepare a no-op microinstruction for subsequent modification by ModifyField(..) and execution by FieldLoop.

ModifyField(Bit1,NBits,Value)
: Modifies the selected microinstruction field.

xxFIELDASM.ASM (Field overlay)

NFields contains the number of fields in a microinstruction

FieldTab
: a table of self-relative pointers to field names indexed by FieldX (0 to NFields-1 are indices)

FMaskTab

FBit1Tab

xxPESCAN.BCPL (PEscan overlay)

ScanForPE(MemX,WaitP,NPETab)
: Scans memory MemX for parity errors. NPETab is a vector in which a summary of results is reported. NPETab!0 is the displacement to the next unused word of this vector. The items in NPETab are three words long, with word 0 accumulating the error count and words 1 and 2 pointing at strings that are printed after the error count. Hence, if the memory has only one parity error, it would set words 1 and 2 to point at " in " and MEMNAM!MemX, and these will get printed as part of the summary. If the memory has several parity error bits, it will use several different 3-word NPETab entries. NPETab!0 is incremented past the words that are used. Calls ReportPE for each error.

xxVM.BCPL (resident code for managing virtual control store and fast symbol search)

LookUpAA(VA,DVec)

Returns the absolute address AA corresponding to a virtual address VA; returns #137777 when there is no corresponding AA. When the optional DVec argument is provided, the AA field in that DVec is filled in with the AA.

LookUpVA(AA,lvPtr)

Returns the VA corresponding to AA or #137777 if no corresponding VA; if the optional lvPtr arg is passed with the call, it is filled in with a pointer to the AAtab table entry, so that other information associated with the VA can be obtained (irrelevant if no corresponding VA).

SetVirtP(Flag,nil,nil)

Called both as an action ("Virt" and "Abs") and directly; Flag is true to enter virtual mode, false to enter absolute mode. The second and third arguments are passed only when SetVirtP is called as an action; their presence causes UpdateMPDValues and FormCmdMenu to be called.

RetrieveBlock(TablePtr,Kind,Strategy,Block,InitVal)

Used to create and retrieve record file blocks associated with the VA_AA, AA_VA, and FastSearch operations. TablePtr!0 contains BlockAddr if the block already exists, else 0. If the block exists, GetBlock(TablePtr!0,Kind,Strategy,Block) is called to get the block into core. The InitVal argument is passed with the call only when a new block should be created if no old one exists; in this case, if TablePtr!0 contains 0, MakeNewBlock(NPagesPerStandardBlock,Kind,Strategy,Block) is called to allocate the block on the record file and initialize its BlockTable entry; then all the words in the block are initialized to InitVal. The core address of the block with BlockAddr as its record file index is returned.

FastSearch(Addr,MemX)

is called by SearchBlocks to obtain the symbol record file block containing the largest address symbol in MemX less than or equal to the address Addr (limited to single-precision). FastSearch returns 0 if the answer is not known; otherwise it calls GetBlock to bring the selected block into core and returns the core address.

IMstab, etc.

Tables containing BlockAddr’s for the inverted symbol tables used by FastSearch.

@VirtualP

True in virtual mode, false in absolute mode.

VAtab

Table containing record file BlockAddr’s for the data structure used by LookUpAA.

AAtab

Table containing record file BlockAddr’s for the data structure used by LookUpVA.