BuilduCodeTutorial.bravo

Building a disk for microcode development.

This tutorial will take you stepwise through a procedure for installing the appropriate software on a disk for microcode development. Because this procedure involves two disks and because members of the microcode group utilize parts of this procedure in the daily development environment, it was decided that the several parts of this build procedure should remain disjoint rather than be integrated into a single command file.

There is a document which contains the procedure outlined below in flow-chart form. It is on [Iris]<wd0>BuilduCodeChart.press, .sil. It may be helpful in giving an overview of the process despite its considerable abstractions.

⇒ Be aware that this procedure requires two disks!
⇒ Command files referenced below usually have self-contained documentation.
⇒ Build paths for various modes (Alto, D0, Software-bootable, pushbutton-bootable, etc.) will be addressed at the appropriate points below.

1. Spin up a clean (or eraseable) disk on your Alto. Boot the NetExec and invoke NewOS.boot (see Alto User's Handbook, pp 6). Use the long installation dialog, DO erase the disk, and do not change any parameters. The Alto program host can be Iris, Maxc, etc. and the Alto program directory should be Alto. You do not need a big SysDir.

2. Retrieve and execute [Iris]<wd0>BareBones.cm.
This will install minimal software utilities (Bravo, EmPress, etc.) on your disk for working with microcode. Don't forget to edit the HARDCOPY section of your User.cm and to initialize Bravo before continuing to step three.

3. Retrieve and execute [Iris]<wd0>MidasDisk.cm.
This will install Swat and the Midas loader/debugger on the disk.

4. Retrieve and execute [Iris]<wd0>ExtantFiles.cm.
This will install Micro.run (microcode assembler), MicroD.run (microcode instruction placer), and various microcode files currently needed to build a microcode disk.

5. Retrieve and execute [Iris]<wd0>MicroAll.cm.
This will assemble the entire complement of D0 microcode. It is probably worth getting a hardcopy of this file.

⇒ Should we tell user to check for errors here?

As you are doubtless aware, there are two modes of microcode for the D0: Alto-compatible and D0-compatible. The following prose assumes Alto-mode in normal text and D0-mode in parenthesized text, e.g. the phrase "execute @Alto-Mumble.cm (@D0-Mumble.cm)" implies running Alto-Mumble.cm for an Alto-mode build and D0-Mumble.cm for a D0-mode build.

6. Retrieve and execute [Iris]<wd0>BMesa1.cm (BDMesa1.cm).
This builds the first block of the microcode, BMesa1.mb (BDMesa1.mb).

7. Retrieve and execute [Iris]<wd0>BAMesa.cm (BDMesa.cm).
This builds the second block of the microcode, BAMesa.mb (BDMesa.mb).

8. Check your disk directory for the existence of the microcode binary files that should have been produced by the two preceding steps: BMesa1.mb and BAMesa.mb (BDMesa1.mb and BDMesa.mb). STORE these two files on your local file server and spin down the disk. We will now go to the second disk, since the current one probably has

only a few hundred free pages remaining.

9. Using another clean or eraseable disk, build a Basic Mesa 5.0 disk as follows:

A. Invoke NewOS.boot from the NetExec as detailed in step 1 of this procedure.
B. Retrieve and execute [Iris/Isis]<Mesa>MesaDisk.cm.

This Basic Mesa 5.0 disk will contain Bravo. Other utilities may be added later if free space is available. Don't do it now!

10. Retrieve the following files:

[your-IFS]<your-directory>BMesa1.mb, BAMesa.mb (BDMesa1.mb, BDMesa.mb)
[Iris]<wd0>MakeLoaderFile.bcd

For creating pushbutton-bootable microcode only:
[Iris]<wd0>MakeAltoBoot.mlf (MakeD0Boot.mlf)
[Iris]<wd0>AltoBoot.cm (D0Boot.cm)
[Iris]<Alto>MoveToKeys.run

For creating software-bootable microcode only:
[Iris]<wd0>MakeAsb.mlf (MakeDsb.mlf)

11. If you want software-bootable microcode, this is the last step. If not, skip this step and go to step 12.

For software-bootable microcode only, execute:

MakeLoaderFile.bcd MakeAsb.mlf (MakeLoaderFile.bcd MakeDsb.mlf)

This should produce the software-bootable microcode file named AMesa.sb (DMesa.sb) on your disk. This completes the build procedure. The steps below pertain only to building pushbutton-bootable microcode.

12. If you want pushbutton-bootable microcode, then continue with this step. If you want software-bootable microcode, go back to step 11.

For pushbutton-bootable microcode only, execute:

MakeLoaderFile.bcd MakeAltoBoot.mlf (MakeLoaderFile.bcd MakeD0Boot.mlf)

This should produce the bootable microcode file named AltoBoot.bt (D0Boot.bt). To make this microcode PUSHBUTTON-bootable, execute:

@AltoBoot.cm (@D0Boot.cm)

This disk should now be pushbutton-bootable on a D0. Try it.

13. End of procedure.

```
// [Iris]<wd0>MicroAll.cm
// Last modified by Chang on October 11, 1979  5:32 PM
// modified by Chang on September 7, 1979  1:33 PM
// Modified by Maxion on August 24, 1979

// This command file assembles the entire complement of D0 microcode.
// These commands will produce the .DIB files used by MicroD.
// This procedure requires about 575 disk pages and about 35 min. to run.

// Documentation:
// Self contained;
// [Iris]<wd0>Help-wd0.bravo;
// Micro: Machine-Independent MicroAssembler by Fiala, Deutsch, Lampson.

// Micro assembles a sequence of source files with default extension ".mc" and outputs
// four files whose extensions are ".MB", ".ER", ".LS", and ".ST".  The default name for
// these is the name of the last source file assembled.  Note that applied to this
// command file (MicroAll.cm) there is a hack in the file D0Lang.mc that causes the file
// extension ".MB" to be changed to ".DIB".  The pertinent line from D0Lang.mc is:
//      "BUILTIN[SETMBEXT@,47];     *Set .mb file


// You must have the microcode assembler on your disk
//      [Iris]<wd0>Micro.run     or [Maxc]<Alto>Micro.run

// ... and the global files
//      [Iris]<wd0>GlobalDefs.mc and D0Lang.mc

// ... and the definitions file(s) as necessary for each module.
//      [Iris]<wd0>UIDefs.mc, DMDefs.mc, EtherDefs.mc, XWDefs.mc, OccupiedDefs.mc

// The following is true for the assembly commands below:
//      <FileName> defaults to <FileName.mc>
//      The "/o" means that symbol table (.ST) output will be suppressed.
//      The "/u" means that all source characters will be converted to upper case.
//      The "/b" means the binary will be listed on the file named before the slash.
//      Outputs from Micro.run are as follows:
//          FileName.DIB  - D-machine intermediate binary
//          FileName.ER   - error list
//          FileName.LS   - assembly listing
//          FileName.ST   - symbol table


// IUTFP (Interim User Terminal, Full Page)
// Requires [Iris]<wd0>UIDefs.mc
micro/o/u UIInit
micro/o/u UITask
micro/o/u Key

// UTLF (User Terminal, Full Page)
// Requires [Iris]<wd0>LFDefs.mc
micro/o/u LFInit
micro/o/u LFTask

// IRDC (Interim Rigid Disk Controller)
// Requires [Iris]<wd0>DMDefs.mc
micro/o/u DMInit
micro/o/u DMTask

// RDC (Rigid Disk Controller)
micro/o/u RDC

// EtherNet + RS232
// Requires [Iris]<wd0>EtherDefs.mc
micro/o/u EtherInit
micro/o/u EtherTask
micro/o/u rs232Sio

// EtherNet masquerading as XWire
// Requires [Iris]<wd0>XWDefs.mc
micro/o/u XWInit
micro/o/u XWTask
micro/o/u XWSio2

// Nova emulator
micro/o/u Nova

// Mesa emulator, D0 mode
micro/o/u MesaLS
micro/o/u MesaJ
micro/o/u MesaX
micro/o/u MesaP

// Mesa emulator, Alto mode
micro/o/u aMesaLS/b AltoMode MesaLS
micro/o/u aMesaJ/b AltoMode MesaJ
micro/o/u aMesaX/b AltoMode MesaX

// BitBlt (Bit Block Transfer), both Alto and D0 modes
micro/o/u BitBlt
micro/o/u aBitBlt/b AltoMode BitBlt

// Initialization: devices and memory
micro/o/u Initialize
micro/o/u AInitialize/b AltoMode Initialize

// Fault handler
micro/o/u Fault

// Dead start and timer code
micro/o/u Timer
```

```
// Two Stage Overlay code
micro/o/u Overlay

// Blocks I and II reserved words
// Requires [Iris]<wd0>OccupiedDefs.mc
micro/o/u Mesa1occupied
micro/o/u Mesa2occupied
```

```
// [Iris]<wd0>AltoBoot.cm
// Last modified on August 17, 1979
// Last editor was Maxion.

// Command file to make a NEW Alto-mode, D0-bootable disk.

// Executing this file in the presence of the files listed below
// will make your disk bootable from the D0 Maintenance Panel.
// This procedure does not require free disk pages to run.

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// This command file requires:
//   [Iris]<Alto>MoveToKeys.run
//   [Iris]<wd0>AltoBoot.bt UNLESS a new AltoBoot.bt is generated.
//       (See [Iris]<wd0>Help-wd0.bravo.)

MoveToKeys AltoBoot.bt 56E

// This disk is now bootable from a D0 Maintenance Panel.
```

```
// [Iris]<wd0>AMesa.cm
// Last modified on 28 August 1979.
// Last editor was Maxion.

// Builds midas boot version, alto mode

// This file is basically undocumented, but will be improved later.
// (I'm not even sure what it's for or if it works. - RM)

MicroD/c AMesa/o Nova UIInit UITask Key DMInit DMTask EtherInit EtherTask rs232SIO aMesaLS aMesaJ aMesaX MesaP BitBlt Fault mInitialize
```

```
// [Iris]<wd0>BAMesa.cm
// Last modified on August 17, 1979
// Last editor was Maxion.

// Builds second block of D0 microcode, Alto-mode.
// Generates BAMesa.mb, which can be loaded into a D-machine and run by Midas.
// This procedure requires about 90 free disk pages.

// Documentation:
// [Iris]<wd0>Help-wd0.bravo
// MicroD Manual by Peter Deutsch
// For origins of AMesaLS, etc. and other ".dib" files see [Iris]<wd0>MicroAll.cm
// The ".dib" files are output from the MicroAssembler, Micro (Micro.run).

// This command file requires the following files on your disk:
//     [Iris]<wd0>MicroD.run
//     and the .dib files (default extension) named in the commands below.

// The result of running this command file will be the generation of:
//     BAMesa.MB
//           ~.csmap for each file.          Invoked by the global "/m" switch.
//           ~.occupied for each file.       Invoked by the global "/m" switch.
//           ~.regs for each file.           Invoked by the global "/m" switch.

MicroD/m/o/r BAMesa/o Mesa1Occupied Nova AMesaLS AMesaJ AMesaX MesaP ABitBlt
```

```
// [Iris]<wd0>BareBones.cm
// Last modified on August 20, 1979.
// Last editor was Maxion.

// This command file will install the minimum (more or less) software on a disk to be
// used for microcode (or other) development.  The user should have "primed" a new
// disk by booting NewOS.BOOT from the NetExec before using this command file.
// This procedure requires about 1200 free disk pages.

// This file will install:
//      Bravo           for editing microcode, etc.
//      Empress         for printing with ease
//      Neptune         for perusing your directory
//      Swat            for Midas ... and other things ... to work

// Any of these may be deleted except Swat, without which Midas will give you grief.
// Removal of any of the others will only inconvenience you.

// Documentation:
// Self contained.

Login

// Retrieve tools and fonts
Ftp.run Maxc ↑
↑
Directory/c Alto Retrieve/c RetrieveBravo.cm EmPress.run Neptune.run InstallSwat.run ↑
↑
Retrieve/s NProg-User.cm User.cm ↑
↑
Directory/c AltoFonts Retrieve/c TimesRoman8.al TimesRoman10.al TimesRoman12.al TimesRoman14.al Helvetica8.al Helvetica10.al Helvetica12.
**al Helvetica12b.al Helvetica18.al Gacha8.al Gacha10.al Gacha12.al Hippo10.al Logo24.al Math10.al Arrows10.al ↑
↑
Directory/c Fonts Retrieve/c Fonts.Widths

// Install Swat
InstallSwat
Delete InstallSwat.run

// Bravo is not fetched until here because installing it requires the font
// files and User.cm.  Be sure to re-install Bravo if you change user.cm.

@RetrieveBravo.cm

Delete RetrieveBravo.cm Dumper.Boot DMT.boot

// Install a variable pitch system font
Copy SysFont.al ← TimesRoman12.al

// Must boot now because of OS bug that may cause it to
// crash if SysFont.al got bigger.

BootFrom Sys.Boot


// Please edit the [HARDCOPY] section in User.cm to specify the name
// of your regular Press-printing server, then say Bravo/i.

// Delete this command file.
Delete BareBones.cm
```

```
// [Iris]<wd0>BDMesa.cm
// Last modified on August 17, 1979
// Last editor was Maxion.

// Builds second block of D0 microcode, D0-mode.
// Generates BDMesa.mb, which can be loaded into a D-machine and run by Midas.
// This procedure requires about 90 free disk pages.

// Documentation:
// [Iris]<wd0>Help-wd0.bravo
// MicroD Manual by Peter Deutsch
// For origins of ".dib" files see [Iris]<wd0>MicroAll.cm
// The ".dib" files are output from the MicroAssembler, Micro (Micro.run).

// This command file requires the following files on your disk:
//      [Iris]<wd0>MicroD.run
//      and the .dib files (default extension) named in the commands below.

// The result of running this command file will be the generation of:
//      BDMesa.MB
//           ~.csmap for each file.          Invoked by the global "/m" switch.
//           ~.occupied for each file.       Invoked by the global "/m" switch.
//           ~.regs for each file.           Invoked by the global "/m" switch.

MicroD/m/o/r BDMesa/o Mesaloccupied MesaLS MesaJ MesaX MesaP BitBlt
```

```
// [Iris]<wd0>BDMesa1.cm
// Last modified by Chang on October 10, 1979  5:42 PM
// modified by Chang on September 7, 1979  1:25 PM
// Modified by Maxion on August 24, 1979

// Builds first block of D0 microcode, D0-mode.
// Generates BDMesa1.mb, which can be loaded into a D-machine and run by Midas.
// This procedure requires about 120 free disk pages.

// Documentation:
// [Iris]<wd0>Help-wd0.bravo
// MicroD Manual by Peter Deutsch
// For origins of ".dib" files see [Iris]<wd0>MicroAll.cm
// The ".dib" files are output from the MicroAssembler, Micro (Micro.run).

// This command file requires the following files on your disk:
//      [Iris]<wd0>MicroD.run
//      and the .dib files (default extension) named in the commands below.


// The result of running this command file will be the generation of:
//      BDMesa1.MB
//              ~.csmap for each file.          Invoked by the global "/m" switch.
//              ~.occupied for each file.       Invoked by the global "/m" switch.
//              ~.regs for each file.           Invoked by the global "/m" switch.

MicroD/m/o/r BDMesa1/o Mesa2occupied LFInit LFTask Key DMInit DMTask RDC XWInit XWSio2 XWTask rs232Sio Fault Initialize Overlay Timer
```

```
// [Iris]<wd0>BMesa1.cm
// Last modified by Chang on September 12, 1979  9:19 AM
// Modified by Maxion on August 24, 1979

// Builds first block of D0 microcode, Alto-mode.
// Generates BMesa1.mb, which can be loaded into a D-machine and run by Midas.
// This procedure requires about 125 free disk pages.

// Documentation:
// [Iris]<wd0>Help-wd0.bravo
// MicroD Manual by Peter Deutsch
// For origins of ".dib" files see [Iris]<wd0>MicroAll.cm
// The ".dib" files are output from the MicroAssembler, Micro (Micro.run).

// This command file requires the following files on your disk:
//     [Iris]<wd0>MicroD.run
//     and the .dib files (default extension) named in the commands below.

// The result of running this command file will be the generation of:
//     BMesa1.MB
//          ~.csmap for each file.          Invoked by the global "/m" switch.
//          ~.occupied for each file.       Invoked by the global "/m" switch.
//          ~.regs for each file.            Invoked by the global "/m" switch.

MicroD/m/o/r BMesa1/o Mesa2occupied LFInit LFTask Key DMInit DMTask RDC EtherInit EtherTask rs232Sio Fault AInitialize Overlay Timer
```

```
// [Iris]<wd0>BuildSys.cm
// Last modified on August 17, 1979
// Last editor was Maxion.

// This command file builds the D0 microcode system.
//      It is essentially the concatenation of the ".cm" files in the commands below PLUS
//      the apparatus to create the pushbutton & soft boot files for Alto- and D0-modes.

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// Salient results:
// Create the pushbutton & soft boot files for Alto- and D0-modes.
// These files will be named AltoBoot.bt, AMesa.sb, D0Boot.bt, and DMesa.sb
// where ".bt" indicates pushbutton-bootable and ".sb" indicates software-bootable.

// This command file requires the following files on your disk:
//      [Iris]<wd0>MicroD.run
//      [Iris]<wd0>MakeLoaderFile.bcd
//      [Iris]<wd0>MakeAll.mlf
//      ... and all the appropriate ".dib" files required for the ".cm" files below.
// It is usual to have run MicroAll.cm before running this file.

// Build first block of D0 microcode, Alto-mode: BMesa1.mb.
@BMesa1.cm
delete BMesa1.dls

// Build second block of D0 microcode, Alto-mode: BAMesa.mb.
@BAMesa.cm
delete BAMesa.dls

// Build first block of D0 microcode, D0-mode: BDMesa1.mb.
@BDMesa1.cm
delete BDMesa1.dls

// Build second block of D0 microcode, D0-mode: BDMesa.mb.
@BDMesa.cm
delete BDMesa.dls

// Create the pushbutton & soft boot files for Alto- and D0-modes.
MakeLoaderFile MakeAll.mlf
```

```
EMPRESS.RUN Gacha/f 6/p daisy/h AltoMode.mc BAMesaOccupied.mc BDMesaloccupied.mc BDMesaOccupied.mc BitBlt.mc BMesaloccupied.mc DOLang.mc
**DMDefs.mc DMInit.mc DMTask.mc EtherDefs.mc EtherInit.mc EtherTask.mc Fault.mc GlobalDefs.mc Initialize.mc Kernel.mc Key.mc LFdefs.mc LF
**init.mc LFkey.mc LFtask.mc Mesaloccupied.mc Mesa2occupied.mc MesaJ.mc MesaLS.mc MesaP.mc MesaX.mc Nova.mc OccupiedDefs.mc Overlay.mc RD
**C.mc RDCdefs.mc rs232Async.mc rs232AsyncTest.mc rs232Bit.mc rs232BitTest.mc rs232Byte.mc rs232ByteTest.mc rs232Defs.mc rs232occupied.mc
** RS232SIO.mc rs232Test.mc Timer.mc UIDefs.mc UIInit.mc UITask.mc XWDefs.mc XWInit.mc XWInit2.mc XWSio.mc XWSio2.mc XWTask.mc AltoBoot.c
**m AMesa.cm BAMesa.cm BareBones.cm BDMesa.cm BDMesa1.cm BMesa1.cm BuildSys.cm Com.Cm D0Boot.cm DMesa.cm ExtantFiles.cm line.cm MicroAll.
**cm MidasDisk.cm Rem.Cm rs232BuildSB.cm rs232Test.cm User.cm
```

```
// [Iris]<wd0>D0boot.cm
// Last modified on August 17, 1979
// Last editor was Maxion.

// Command file to make a NEW D0-mode, D0-bootable disk.

// Executing this file in the presence of the files listed below
// will make your disk bootable from the D0 Maintenance Panel.
// This procedure does not require free disk pages to run.

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// This command file requires:
//    [Iris]<Alto>MoveToKeys.run
//    [Iris]<wd0>D0boot.bt UNLESS a new D0boot.bt is generated.
//        (See [Iris]<wd0>Help-wd0.bravo.)

MoveToKeys D0boot.bt 56E

// This disk is now bootable from a D0 Maintenance Panel.
```

```
// [Iris]<wd0>DMesa.cm
// Last modified on 28 August 1979.
// Last editor was Maxion.

// Builds midas boot version, D0 mode

// This file is basically undocumented, but will be improved later.
// (I'm not even sure what it's for or if it works. - RM)

MicroD/c DMesa/o Nova UIInit UITask Key DMInit DMTask EtherInit EtherTask rs232Sio MesaLS MesaJ MesaX MesaP BitBlt Fault mInitialize
```

```
// [Iris]<wd0>ExtantFiles.cm
// Last modified Chang, October 17, 1979  11:32 AM
// modified Chang, September 28, 1979  5:34 PM
// Modified by Maxion on August 24, 1979

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// This command file will fetch and install:
//      the microcode files currently needed to build a microcode disk from scratch;
//      the microcode assembler, Micro.run:
//      the microcode instruction placer, MicroD.run.

// This procedure requires about 1000 free disk pages.


// Brief explanation of file requirements for Microcode:

// Micro.run  ...  microcode assembler - needed for MicroAll.cm
// MicroD.run  ...  microcode instruction placer - needed for building Mesa microcode
// D0Lang.mc  ...  microcode definition language - needed for MicroAll.cm
// GlobalDefs.mc  ...  global definitions for microcode - needed for MicroAll.cm
// ~.mlf  ...  data for MakeLoaderFile.

// For brief explanations of the rest of these microcode files, see MicroAll.cm.

FTP Iris Directory/c <wd0> Retrieve/c Micro.run MicroD.run GlobalDefs.mc D0Lang.mc ↑
AltoMode.mc MesaLS.mc MesaJ.mc MesaX.mc MesaP.mc BitBlt.mc Nova.mc Initialize.mc Fault.mc UIDefs.mc UIInit.mc UITask.mc LFDefs.mc LFInit.
**mc LFTask.mc XWdefs.mc XWInit.mc  XWSio2.mc XWTask.mc Key.mc DMDefs.mc DMInit.mc DMTask.mc RDCdefs.mc RDC.mc EtherDefs.mc EtherInit.mc
**EtherTask.mc RS232SIO.mc Timer.mc Overlay.mc Mesa1Occupied.mc Mesa2Occupied.mc OccupiedDefs.mc Kernel.mc ↑
↑
MicroAll.cm BMesa1.cm BDMesa1.cm BAMesa.cm BDMesa.cm
```

```
ftp iris dir/c <ad0>40w> ret/c AltoMode.mc BAMesaOccupied.mc BDMesa1occupied.mc BDMesaOccupied.mc BitBlt.mc BMesa1occupied.mc D0Lang.mc D
**MDefs.mc DMInit.mc DMTask.mc EtherDefs.mc EtherInit.mc EtherTask.mc Fault.mc GlobalDefs.mc Initialize.mc Kernel.mc Key.mc LFdefs.mc LFi
**nit.mc LFkey.mc LFtask.mc Mesa1occupied.mc Mesa2occupied.mc MesaJ.mc MesaLS.mc MesaP.mc MesaX.mc Nova.mc OccupiedDefs.mc Overlay.mc RDC
**.mc RDCdefs.mc rs232Async.mc rs232AsyncTest.mc rs232Bit.mc rs232BitTest.mc rs232Byte.mc rs232ByteTest.mc rs232Defs.mc rs232occupied.mc
**RS232SIO.mc rs232Test.mc Timer.mc UIDefs.mc UIInit.mc UITask.mc XWDefs.mc XWInit.mc XWInit2.mc XWSio.mc XWSio2.mc XWTask.mc AltoBoot.cm
** AMesa.cm BAMesa.cm BareBones.cm BDMesa.cm BDMesa1.cm BMesa1.cm BuildSys.cm D0Boot.cm DMesa.cm ExtantFiles.cm MicroAll.cm MidasDisk.cm
**rs232BuildSB.cm rs232Test.cm

empress Gacha6/f AltoMode.mc BAMesaOccupied.mc BDMesa1occupied.mc BDMesaOccupied.mc BitBlt.mc BMesa1occupied.mc D0Lang.mc DMDefs.mc DMIni
**t.mc DMTask.mc EtherDefs.mc EtherInit.mc EtherTask.mc Fault.mc GlobalDefs.mc Initialize.mc Kernel.mc Key.mc LFdefs.mc LFinit.mc LFkey.m
**c LFtask.mc Mesa1occupied.mc Mesa2occupied.mc MesaJ.mc MesaLS.mc MesaP.mc MesaX.mc Nova.mc OccupiedDefs.mc Overlay.mc RDC.mc RDCdefs.mc
** rs232Async.mc rs232AsyncTest.mc rs232Bit.mc rs232BitTest.mc rs232Byte.mc rs232ByteTest.mc rs232Defs.mc rs232occupied.mc RS232SIO.mc rs
**232Test.mc Timer.mc UIDefs.mc UIInit.mc UITask.mc XWDefs.mc XWInit.mc XWInit2.mc XWSio.mc XWSio2.mc XWTask.mc AltoBoot.cm AMesa.cm BAMe
**sa.cm BareBones.cm BDMesa.cm BDMesa1.cm BMesa1.cm BuildSys.cm D0Boot.cm DMesa.cm ExtantFiles.cm MicroAll.cm MidasDisk.cm rs232BuildSB.c
**m rs232Test.cm
```

```
// [Iris]<wd0>MidasDisk.cm
// Last modified on August 20, 1979
// Last editor was Maxion.

// This command file:
//  will (re)install Midas (loader/debugger) on your fresh (or present) disk;
//  will (re)install Swat on your fresh (or present) disk;
//    (Swat is helpful, but not essential.)
//  will handle ONLY Midas and Swat - nothing more!!
//  requires about 510 disk pages to run.

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// Get Swat and install it.  Also get appropriate Midas files.

FTP Iris Directory/c <Alto> Retrieve/c InstallSwat.run ↑
↑
Directory/c <WD0> Retrieve/c AMesa.midas BAMesa.mb BDMesa.mb BDMesa1.mb BMesa1.mb Boot.midas DMesa.midas D0Loader.mb Kernel.mb Midas.mida
**s Midas.programs Midas.run User-Midas.programs

InstallSwat
Delete InstallSwat.run

Delete MidasDisk.cm

// Initialize Midas.
Midas/i


// Delete Dumper.boot if you wish.
// Delete DMT.boot if you wish.

// Swat users:  Consider obtaining the new file Swat.help.
```

```
// [Iris]<wd0>rs232BuildSB.cm
// Last modified on September 18, 1979
// Last editor was Maxion.

// This command file builds software-bootable files for rs232 microcode.

// Salient outputs from this command file are:
//     rs232Async.sb
//     rs232Byte.sb
//     rs232Bit.sb

// Documentation:
// Self contained and [Iris]<wd0>Help-wd0.bravo

// This command file requires the following files on your disk:
//     [Iris]<wd0>Micro.run   or   [Maxc]<Alto>Micro.run    (the microcode assembler)
//     [Iris]<wd0>MicroD.run
//     [Iris]<wd0>MakeLoaderFile.bcd
//     [Iris]<wd0>D0Lang.mc
//     [Iris]<wd0>rs232Defs.mc
//     and the .mc files (default extension) and .mlf files named in the commands below.

// Assemble rs232occupied.mc
Micro/u/o rs232occupied

// Build Asynchronous system: software-bootable file rs232Async.sb
Micro/u/o rs232Async
MicroD/n rs232Async/o rs232occupied rs232Async
MakeLoaderFile rs232Async.mlf

// Build Byte Synchronous system: software-bootable file rs232Byte.sb
Micro/u/o rs232Byte
MicroD/n rs232Byte/o rs232occupied rs232Byte
MakeLoaderFile rs232Byte.mlf

// Build Bit Synchronous system: software-bootable file rs232Bit.sb
Micro/u/o rs232Bit
MicroD/n rs232Bit/o rs232occupied rs232Bit
MakeLoaderFile rs232Bit.mlf
```

```
// [Iris]<wd0>rs232Test.cm
// Last modified on June 1, 1979
// Last editor was BRD.

// This command file builds RS232 microcode test programs.

// Salient output files from this command file are the microcode binaries:
//     rs232AsyncTest.mb
//     rs232ByteTest.mb
//     rs232BitTest.mb

// Documentation:
// Micro: Machine-Independent MicroAssembler by Fiala, Deutsch, Lampson
// MicroD Manual by Peter Deutsch

// This command file requires the following files on your disk:
//     [Iris]<wd0>Micro.run   or   [Maxc]<Alto>Micro.run    (the microcode assembler)
//     [Iris]<wd0>MicroD.run
//     [Iris]<wd0>rs232Async.mc        for asynchronous test only
//     [Iris]<wd0>rs232Byte.mc          for byte asynchronous test only
//     [Iris]<wd0>rs232Bit.mc           for bit asynchronous test only
//     [Iris]<wd0>rs232Test.mc          for all tests
//     ... and the .mc files (default extension) named in the commands below.

// Build Asynchronous Test
Micro/u/o rs232AsyncTest
MicroD/n rs232AsyncTest

// Build Byte Synchronous Test
Micro/u/o rs232ByteTest
MicroD/n rs232ByteTest

// Build Bit Synchronous Test
Micro/u/o rs232BitTest
MicroD/n rs232BitTest
```

```
[BRAVO]
A.INIT:"{6,2,0,4}g'@1
**@@G[@1]{6,2,0,0}l;*lz@@L[177762,177777,]756
***i'
**Page Numbers: Yes First Page: 1
Heading:
***l;*h
*q
y@@E"

B.INIT:"{6,2,0,4}g'@1config
**@@G[@1config]{6,2,0,0}wny1114,y1114,y1114,{6,4,0,4}g'@1errlog
**@@G[@1errlog]{6,2,0,0}n@@E"

C.INIT:"{6,1,0,0}g'line.cm
**@@G[line.cm]@@E"

D.INIT:"{6,2,0,0}g'@1
**@@G[@1]{6,2,0,0}wny660,y660,y660,{6,4,0,0}g'@2
**@@G[@2]@@E"

G.INIT:"{6,2,0,6}g'@1
**@@G[@1]{6,2,0,0}h
*Q
@@E"

H.INIT:"{6,2,0,6}g'@1
**@@G[@1]{6,2,0,0}hc'@2
**@@@@'@3
**
*q
@@E"

J.INIT:"{6,2,0,4}g'@1
**@@G[@1]{6,2,@2,@2}N@@E"

L.INIT:"{6,2,0,4}g'@1config
**@@G[@1config]{6,2,0,0}wny1114,y1114,y1114,{6,4,0,4}g'@1errlog
**@@G[@1errlog]{6,2,0,0}@@E"

M.INIT:"{6,2,0,4}g'@1mesa
**@@G[@1mesa]{6,2,0,0}wny1114,y1114,y1114,{6,4,0,4}g'@1errlog
**@@G[@1errlog]{6,2,0,0}@@E"

N.INIT:"{6,1,0,0}g'@1
**@@G[@1]@@E"

S.INIT:"{6,2,0,4}g'@1.mesa
**@@G[@1.mesa]{6,2,0,0}l;*lz@@L[177762,177777,]756
***i'
**Page Numbers: Yes First Page: 1
Heading:
***l;*h
*q
y@@E"

A.QUIT:"{6,2,0,0}l;*lz@@L[177762,177777,]756
***i'
**Page Numbers: Yes First Page: 1
Heading:
***l;*h
*q
"

B.QUIT:"{6,1,0,0}q
Delete @4errlog; Binder /-p @4/g IF.run/r @4errlog then Bravo/B @4
"

C.QUIT:"*q
@@@4@@
"

D.QUIT:"{6,2,0,0}q
BRAVO/D @4 @5
"

F.QUIT:"*q
FTP Iris Store/c @4; Delete @4 @4$
"

G.QUIT:"*q
Chat @4/D
"

M.QUIT:"{6,1,0,0}q
Delete @4errlog; Compiler -Alto/c -pause/c @4  IF.run/r @4errlog then Bravo/M @4
"

N.QUIT:"{6,1,0,0}q
Delete @4errlog @5errlog; Compiler -Alto/c -pause/c @4 @5; Bravo/M @4; Bravo/M @5
"

P.QUIT:"{6,1,0,0}q
Delete @4errlog; Compiler -pause/c @4; Bravo/M @4
"

Q.QUIT:"{6,1,0,0}q
Delete @4errlog @5errlog; Compiler -pause/c @4 @5;Bravo/M @4; Bravo/M @5
"

R.QUIT:"{6,2,0,0}q
```

```
BRAVO/N @4
"

S.QUIT:"*q
FTP Iris Store/c @4
"

T.QUIT:"*q
FTP Iris store/s @4 @3>@4
"

U.QUIT:"{6,1,0,0}q
y
Copy temp.ts ← bravo.ts;BRAVO/d user.cm temp.ts
"


FONT:0 HELVETICA 8 HELVETICA 10 HELVETICA 8
FONT:1 HELVETICA 7 HELVETICA 8 HELVETICA 7
FONT:5 HELVETICA 12 HELVETICA 12 HELVETICA 12

OFFSET: Standard offset = 4
TABS: Standard tab width = 635
LEAD: Line leading = 6, Paragraph leading = 12
NESTED: Delta left = 635, Delta right = 0
SCREEN: Screen top = 25, System window end = 90, Screen bottom = 780
MARGINS: Paragraph margin = 2998, Left margin = 2998, Right margin = 20598

[HARDCOPY]
HOST: Iris
PREFERREDFORMAT: PRESS
PRESS: Daisy
PRINTEDBY: "DaleKnutsen"
EXTENSION: .mesa
FONT: HELVETICA 10 MRR

[CHAT]
BELL: DING FLASH
BORDER: BLACK
FONT: Gacha10.al
LINEFEEDS: OFF
TYPESCRIPT: Chat.ts 10000
TYPESCRIPTCHARS: OFF ON

[DDS]
FONT: Helvetica10.al
SMALLFONT: sysfont.al
CONTEXT: not (*.al or *.run or *.image or sys* or *.cm* or *.scratch* or dds* or swat* or bravo.*)
FULLINIT: NO
SELSPEC: *
SORT BY: name, extension
SHOW: size

[TOOLS]
BITMAP: [x: 64, y: 128, w: 478, h: 400]
DEBUG: No
ToolsFont: SysFont.al
FileHost: Iris
FileDirectory: <Wpilot>

[Librarian]
Server: Marion
Root: <CoPilot>Root
NamePrefix: <CoPilot>
NameSuffix: mesa
BcdBucket: [Iris]<Wpilot>
DefaultContentsLocation: [Iris]<Wpilot>
Limpets: pointsto
Children: comprises <*>
Levels: 2
LevelSpacing: 32
InLevelSpacing: 16
NameThru: Yes
Sideways: True

[MAILCHECK]
HOST: Maxc
NEWMAIL: CHAT @H MSG.DO/D
TIME: YES

[EXECUTIVE]
Font: Helvetica12
eventBooted: Login // Hi, Dale!
eventRFC: FTP // eventRFC
eventInstall: // eventInstall
eventAboutToDie: // eventAboutToDie
eventUnknown: // eventUnknown
eventClockWrong: SetTime // eventClockWrong
```

```
*Last Modified by Sandman on February 23, 1979  11:36 AM
BUILTIN[SET,11];
SET[AltoMode, 1];
```

```
* MicroD 8.11 (OS 16) of July 1, 1979
*   at 17-Oct-79 11:59:10

INSERT[OccupiedDefs];

        TITLE[BAMesaOccupied];

* Locations reserved on page 0

        IMRESERVE[0, 2, 75];
        IMRESERVE[0, 156, 10];
        IMRESERVE[0, 173, 4];
        IMRESERVE[0, 300, 32];

* Locations reserved on page 1

        IMRESERVE[1, 0, 400];

* Locations reserved on page 4

        IMRESERVE[4, 0, 340];
        IMRESERVE[4, 341, 1];
        IMRESERVE[4, 345, 1];
        IMRESERVE[4, 351, 1];
        IMRESERVE[4, 355, 1];
        IMRESERVE[4, 361, 1];
        IMRESERVE[4, 365, 1];
        IMRESERVE[4, 371, 1];
        IMRESERVE[4, 375, 1];
        IMRESERVE[4, 377, 1];

* Locations reserved on page 5

        IMRESERVE[5, 0, 400];

* Locations reserved on page 6

        IMRESERVE[6, 0, 363];
        IMRESERVE[6, 365, 1];
        IMRESERVE[6, 371, 1];
        IMRESERVE[6, 375, 1];
        IMRESERVE[6, 377, 1];

* Locations reserved on page 7

        IMRESERVE[7, 0, 400];

* Locations reserved on page 11B

        IMRESERVE[11, 0, 366];

* Locations reserved on page 14B

        IMRESERVE[14, 137, 241];

* Locations reserved on page 15B

        IMRESERVE[15, 0, 376];
        IMRESERVE[15, 377, 1];

* Locations reserved on page 16B

        IMRESERVE[16, 0, 357];

        END;
```

```
* MicroD 8.11 (OS 16) of July 1, 1979
*   at 17-Oct-79 11:09:37

INSERT[OccupiedDefs];

        TITLE[BDMesa1Occupied];

* Locations reserved on page 0

        IMRESERVE[0, 100, 233];
        IMRESERVE[0, 335, 23];
        IMRESERVE[0, 361, 17];

* Locations reserved on page 1

        IMRESERVE[1, 100, 266];

* Locations reserved on page 2

        IMRESERVE[2, 100, 242];
        IMRESERVE[2, 372, 1];
        IMRESERVE[2, 374, 1];
        IMRESERVE[2, 376, 1];

* Locations reserved on page 3

        IMRESERVE[3, 0, 307];
        IMRESERVE[3, 370, 3];
        IMRESERVE[3, 374, 3];

* Locations reserved on page 7

        IMRESERVE[7, 27, 1];
        IMRESERVE[7, 76, 1];

* Locations reserved on page 10B

        IMRESERVE[10, 0, 363];

* Locations reserved on page 12B

        IMRESERVE[12, 0, 375];

* Locations reserved on page 13B

        IMRESERVE[13, 0, 305];

* Locations reserved on page 14B

        IMRESERVE[14, 0, 137];

* Locations reserved on page 15B

        IMRESERVE[15, 300, 1];

* Locations reserved on page 16B

        IMRESERVE[16, 0, 320];

* Locations reserved on page 17B

        IMRESERVE[17, 140, 40];

        END;
```

```
* MicroD 8.11 (OS 16) of July 1, 1979
*   at 17-Oct-79 11:11:10

INSERT[OccupiedDefs];

        TITLE[BDMesaOccupied];

* Locations reserved on page 0

        IMRESERVE[0, 2, 22];
        IMRESERVE[0, 156, 10];
        IMRESERVE[0, 173, 4];
        IMRESERVE[0, 300, 32];

* Locations reserved on page 4

        IMRESERVE[4, 0, 332];
        IMRESERVE[4, 335, 1];
        IMRESERVE[4, 341, 1];
        IMRESERVE[4, 345, 1];
        IMRESERVE[4, 351, 1];
        IMRESERVE[4, 355, 1];
        IMRESERVE[4, 361, 1];
        IMRESERVE[4, 365, 1];
        IMRESERVE[4, 371, 1];
        IMRESERVE[4, 375, 1];
        IMRESERVE[4, 377, 1];

* Locations reserved on page 5

        IMRESERVE[5, 0, 400];

* Locations reserved on page 6

        IMRESERVE[6, 0, 324];
        IMRESERVE[6, 325, 2];
        IMRESERVE[6, 331, 3];
        IMRESERVE[6, 335, 3];
        IMRESERVE[6, 341, 3];
        IMRESERVE[6, 345, 3];
        IMRESERVE[6, 351, 3];
        IMRESERVE[6, 355, 3];
        IMRESERVE[6, 361, 3];
        IMRESERVE[6, 365, 1];
        IMRESERVE[6, 371, 1];
        IMRESERVE[6, 375, 1];
        IMRESERVE[6, 377, 1];

* Locations reserved on page 7

        IMRESERVE[7, 0, 360];
        IMRESERVE[7, 361, 1];
        IMRESERVE[7, 365, 1];
        IMRESERVE[7, 371, 1];
        IMRESERVE[7, 375, 1];
        IMRESERVE[7, 377, 1];

* Locations reserved on page 11B

        IMRESERVE[11, 0, 316];

* Locations reserved on page 14B

        IMRESERVE[14, 137, 233];

* Locations reserved on page 15B

        IMRESERVE[15, 0, 372];
        IMRESERVE[15, 377, 1];

* Locations reserved on page 16B

        IMRESERVE[16, 0, 357];

        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        TITLE[BITBLT];
%
* Last modified by Chang, September 11, 1979  8:46 AM, interrupts off problems
*  modified by Johnsson, June 28, 1979  8:50 AM
*       BBTable format
*
* WORD    NAME
*    0  FUNCTION         bit 0 Long Bitblt; bits 14-17 function (see below)
*    1  unused
*    2  DBCA     Destination BCA   Base Core Address of dest bit map
*    3  DBMR     Destination BMR   Bit Map Width in words
*    4  DLX      Destination LX    Left X offset from first bit
*    5  DTY      Destination TY    Top Y offset from first scan line
*    6  DW       Destination W     Width in bits of bit map
*    7  DH       Destination H     Height in scan lines of bit map
*   10  SBCA     Source BCA
*   11  SBMR     Source BMR
*   12  SLX      Source LX
*   13  STY      Source TY
*   14  Gray0               These four words are the Gray Block
*   15  Gray1          Gray0 is used on the first item,
*   16  Gray2          Gray1 on the second, Gray2 on the third,
*   17  Gray3          Gray3 on the fourth, Gray0 on the fifth,etc.
*   20  LongSourceLo
*   21  LongSourceHi
*   22  LongDestLo
*   23  LongDestHi

*       Bit BLT functions

*       CODE    MA, MB  SALUFOP         Dest ←

*         0      0   0   R OR T          Src
*         1      0   1   R OR T          Src OR Dest
*         2      0   1   R XOR T         Src XOR Dest
*         3      0   1   R AND notT      notSrc AND Dest
*         4      1   0   R OR notT       notSrc
*         5      1   1   R OR notT       notSrc OR Dest
*         6      1   1   R XNOR T        notSrc XOR Dest
*         7      1   1   R AND T         Src AND Dest
*        10      0   0   xxx            (Src AND Gry) OR (notSrc AND Dest)
*        11      1   0   R OR T         (Src AND Gry) OR Dest
*        12      1   0   R XOR T        (Src AND Gry) XOR Dest
*        13      1   0   R AND notT     not(Src AND Gry) AND Dest
*        14      0   0   R OR T          Gry
*        15      1   0   R OR T          Gry OR Dest
*        16      1   0   R XOR T         Gry XOR Dest
*        17      1   0   R AND notT      notGry AND Dest

        interpretation of bbFunction bits

            00   mesa long pointer
            01   mesa called = 1 / nova called = 0
         02-05   unused
            06   bot to top = 1 / top to bot =0
            07   r to l = 1 / l to r = 0
         10-12   which-innerloop index
            13   zero
         14-17   Bitblt function code
%
```

```
*dispatch tables for bitblt
        SET[BBP1P,LSHIFT[BBP1,10]];
        SET[BBP2P,LSHIFT[BBP2,10]];
        SET[BBILA,  ADD[BBP1P,100]];
        SET[BBILB,  ADD[BBP1P,120]];
        SET[BBILC,  ADD[BBP1P,140]];
        SET[BBILD,  ADD[BBP1P,160]];
        SET[BBILE,  ADD[BBP1P,200]];
        SET[BBILDX, ADD[BBP1P,220]];
        SET[bbIdisp,ADD[BBP1P,240]];
        SET[BBF,    ADD[BBP2P,340]];

*       BBFA dispatch values

        set[BBNRM,7];   *no refill
        set[BBDST,6];   *destination refill
        set[BBSRC,5];   *source refill
        set[BBBTH,4];   *source and destination refill
        set[BBITM,3];   *item refill

        set[bbILtype0,    00];
        set[bbILtype1,    02];
        set[bbILtype2,    04];
        set[bbILtype3,    06];
        set[bbILtype4,    10];
```

```
%       Initialization
*       *       determination of bit blt directions
*       (top to bottom , left to right) dty < sty
*       (top to bottom , right to left) (dty = sty) and (dlx > slx)
*       (bottom to top , left to right) ((dty = sty) and (dlx =< slx))
*                                        or (dty > sty)
%

        ONPAGE[BBP2] ;

bbp2ret:        return;

*//******* Start of Alto Code *********************************************
NovaBitBLT:     *AC2,AC3 are a base register pointing to the BitBLT table
        AC0 ← 21c, task;                *Stkp points to ICOM in AC1
        Stkp ← AC0;
        AC0 ← 0c, goto[bbBitBLT]; *AC0 is the "entered from Nova" flag
*//******* End of Alto Code ***********************************************

MesaBitBLT: lu ← xfWDC; *T has address of table
        skip[ALU=0];
        NWW ← (NWW) or (100000c); *disable interrupts in xfWDC#0
        AC2 ← T;
        T ← MDShi, task;
        AC3 ← T;        *Long pointer to BitBLT table in AC2,AC3
        AC0 ← 40000C, goto[bbBitBLT];

*get here when finished, with test of "entered from Mesa" flag pending
bbExit:
*//******* Start of Alto Code *********************************************
        dblgoto[bbMdone,bbNdone,ALU#0];
*//******* End of Alto Code ***********************************************

* bbMdone:      loadpage[4];
*       stack&-2, gotop[MesaBBret];
bbMdone:        stack&-2, loadpage[4];
        NWW ← (NWW) and not (100000C), gotop[MesaBBret];

*//******* Start of Alto Code *********************************************
bbNdone:        loadpage[nePage];
        gotop[neNoskip], FF1@[17];
*//******* End of Alto Code ***********************************************

*common bitblt code
bbBITBLT:
        pfetch1[AC2,bbFunction,0], call[bbp2ret];*fetch function
        pfetch2[AC2,bbRTEMslx,12],task;*fetch slx and sty
        bbSrcQAddrLo ← zero ;
        pfetch2[AC2,bbRTEMdlx,4];*fetch dlx and dty
        t←(17c), task;
        t←(lsh[AC0,1])or(t);
        pfetch2[AC2,bbItemWidth,6], call[bbp2ret];*fetch dw and dh
        bbFunction←(bbFunction)and(t);*Insure no garbage and mask bit 0 if entered from Nova
        t←(AC0);
        task,bbFunction←(bbFunction)or(t); *"called from Mesa" bit
        t ← ldf[bbFunction,14,2] ;
        lu ← bbItemWidth;
        RTEMP← T ,skip[alu#0];
        lu ← ldf[bbFunction,1,1], goto[bbExit]; *Completion return - item width is zero
        RTEMP←t;
        lu ← (RTEMP) xor (3c) ;
        skip[alu#0] ;
        goto[bbTB] , bbSrcQAddrLo ← (1c) ;      *if function is 14-17, source not used,use tblr
        T ← bbRTEMdty;
        LU ← (bbRTEMsty) - (T) ;                        *calc sty - dty
        GOTO[bbBT1,alu<0],freezeresult;
bbA3:   GOTO[bbTB,ALU#0] ;
bbA4:   T ← bbRTEMdlx ;
        LU ← (bbRTEMslx) - (T) ;                        *calc slx - dlx
        DBLGOTO[bbTBRL,bbTBLR,ALU<0] ,t ← Stack;
bbBT1:  GOTO[bbBTLR] ,t ← Stack;        *bottom to top , left to right
bbTB:   GOTO[bbTBLR] ,t ← Stack;        *top to bottom , left to right
```

```
*               for left to right , top to bottom
*               specific initialization
*       sty             sty + icom
bbTBLR:
        bbRTEMsty ← (bbRTEMsty) + (T);
        GOTOp[bbGenlInit],bbRTEMdty ← (bbRTEMdty) + (T) ;
*
*               for left to right , bottom to top
*               specific initialization
bbBTLR:
        task,T ← (bbItemsRemaining) - (T) - 1 ;
        bbRTEMsty ← (bbRTEMsty) + (T) ;
        bbRTEMdty ← (bbRTEMdty) + (T);
        GOTOp[bbGenlInit],bbFunction←(bbFunction)or(1000C);*set L to R bit
*
*               for right to left , top to bottom
*               specific initialization
bbTBRL:
        bbRTEMsty ← (bbRTEMsty) + (T) ;
        task,bbRTEMdty ← (bbRTEMdty) + (T) ;
        t ← bbRTEMslx ;
        t ← (bbRTEMdlx) - (t);
        bbSDNonOverlap ← t ;
        lu ← ldf[bbSDNonOverlap,0,12] ;
        skip[alu#0],bbMinusSDNonOverlap←(zero)-(t);
        goto[bbGenlInit];*L to R if non-overlap < 100b
        lu ← ldf[bbItemWidth,0,12] ;
        skip[alu#0] ;
        goto[bbGenlInit];*L to R if item length < 100b
        lu ← (bbItemWidth) - (t) ;
        skip[carry];
        goto[bbGenlInit];*L to R if item width < non-overlap
        GOTO[bbGenlInit] , bbFunction←(bbFunction)or(400C) ;*R to L
```

```
*               general initialization
*               calc ss
*       ss              (lsh[4](sbca + (sty * sbmr)) + slx)  24 bits

bbGenlInit:
        skip[r even] , lu ← bbSrcQAddrLo ;
        goto[bbinnosrc], pfetch2[AC2,bbDBCA,2];
        pfetch2[AC2,bbSBCA,10] ;*fetch sbca and sbmr
        goto[bbLongSrcGet,r<0],lu←bbFunction;
bbShortSrcGet:
         t←bbSBCA;
         call[bbp2ret],bbSrcQAddrLo←t;
         goto[bbSrcInit],bbSrcQAddrHi←(zero);*short form set-up
bbLongSrcGet:
         t←20c;
         call[bbp2ret],pfetch2[AC2,bbSrcQAddrLo];*long form set-up

*T ← sty * sbmr.  The product is =< 16 bits
bbSrcInit:
        T ← bbSBMR;
        RTEMP ← T;
        T ← 0c, call[bbSrcMul];
bbSrcMul:       RTEMP ← rsh[RTEMP,1], goto[.+2,Reven];
        T ← (bbRTEMsty) + (T);
        bbRTEMsty ← lsh[bbRTEMsty, 1], goto[.+2, ALU=0];
        return;
bbSrcMulDone:
        bbSrcQAddrLo←(bbSrcQAddrLo)+(t);
        skip[nocarry], t←lsh[bbSrcQAddrLo,4];*move SrcQAddr to SrcStartBit
        bbSrcQAddrHi←(bbSrcQAddrHi)+1;
        bbSrcStartBitLo←t;
        t←rsh[bbSrcQAddrLo,14];
        task,t←(lsh[bbSrcQAddrHi,4])+(t);
        bbSrcStartBitHi←t;
        t←bbRTEMslx;*add slx to SrcStartBit
        bbSrcStartBitLo←(bbSrcStartBitLo)+(t);
        skip[nocarry], bbSrcQAddrLo←(bbSrcQAddrLo)and not(3c);
        bbSrcStartBitHi←(bbSrcStartBitHi)+1;* SrcStartBit now complete
*               calc ds
*       ds              (lsh[4](dbca + (dty * dbmr)) + dlx)  24 bits
        pfetch2[AC2,bbDBCA,2];*fetch dbca and dbmr
bbinnosrc:
        goto[bbShortDestGet,r>=0],lu←bbFunction;
bbLongDestGet:
        t←22c;
        goto[bbDestInit],pfetch2[AC2,bbDestQAddrLo];
bbShortDestGet:
        t←bbDBCA;
        task,bbDestQAddrLo←t;
        bbDestQAddrHi←(zero);*short form set-up
*T ← dty * dbmr.  16-bit product
bbDestInit:
        T ← bbDBMR;
        RTEMP ← T;
        T ← 0c, call[bbDestMul];
bbDestMul:      RTEMP ← rsh[RTEMP,1], goto[.+2, Reven];
        T ← (bbRTEMdty) + (T);
        bbRTEMdty ← lsh[bbRTEMdty, 1], goto[.+2, ALU=0];
        return;

bbDestMulDone:
        bbDestQAddrLo←(bbDestQAddrLo)+(t);
        skip[nocarry],t←lsh[bbDestQAddrLo,4];*move DestQAddr to DestStartBit
        bbDestQAddrHi←(bbDestQAddrHi)+1;
        bbDestStartBitLo←t;
        goto[.+1],t←rsh[bbDestQAddrLo,14];
        t←(lsh[bbDestQAddrHi,4])+(t);
        task,bbDestStartBitHi←t;
        t←bbRTEMdlx;*add dlx to DestStartBit
        bbDestStartBitLo←(bbDestStartBitLo)+(t);
        skip[nocarry],t ← bbItemWidth;
        bbDestStartBitHi←(bbDestStartBitHi)+1;* DestStartBit now complete
        bbMinusItemWidth ← (zero) - (T) ;       *want minus item width
*       test for specific initialization (bottom to top)
        lu←ldf[bbFunction,6,1] ;
        goto[bbILX0,alu=0] ;
*                       this will go to bbilx0 if t to b
*                       or bbilx1 if b to t

bbILX1: t ← bbSBMR ;
        task,bbSBMR ← (zero) - (t) ;
        t ← bbDBMR ;
        bbDBMR ← (zero) - (t) ;

bbILX0:
        T ← (Stack);
        bbItemsRemainingMinus1 ← (bbItemsRemaining) - (T) - 1 ;
        skip[alu>=0] , bbItemsRemainingMinus2 ← (bbItemsRemainingMinus1) - 1 ;
        lu ← ldf[bbFunction,1,1], goto[bbExit]; *Completion return - no items remaining
        bbGrayCnt ← T ;
        skip[r even] , lu ← bbSrcQAddrLo;
        goto[.+2] , SB ← bbDestStartBitLo ;
        SB ← bbSrcStartBitLo ;
        task,DB ← bbDestStartBitLo ;
        MNBR ← bbMinusItemWidth;

bbfd:   DISPATCH[bbFunction,14,4] ;
        DISP[bbFUN00] ;
*               will dispatch to bbfunN for function N
```

```
bbFUN00:        goto[bbEndInit] , T ← (204C) , AT[BBF,0] ;*salufop ← [0,0,r or t]
bbFUN01:        goto[bbEndInit] , T ← (304C) , AT[BBF,1] ;*salufop ← [1,0,r or t]
bbFUN02:        goto[bbEndInit] , T ← (363C) , AT[BBF,2] ;*salufop ← [1,0,r xor t]
bbFUN03:        goto[bbEndInit] , T ← (327C) , AT[BBF,3] ;*salufop ← [1,0,r and not t]
bbFUN04:        goto[bbEndInit] , T ← (074C) , AT[BBF,4] ;*salufop ← [0,1,r or not t]
bbFUN05:        goto[bbEndInit] , T ← (104C) , AT[BBF,5] ;*salufop ← [1,1,r or t]
bbFUN06:        goto[bbEndInit] , T ← (154C) , AT[BBF,6] ;*salufop ← [1,1,r xnor t]
bbFUN07:        goto[bbEndInit] , T ← (156C) , AT[BBF,7] ;*salufop ← [1,1,r and t]
bbFUN10:        T ← (200C) , AT[BBF,10] ;                 *salufop ← [x,0,x]
        goto[bbEndInit] , bbFunction ← (bbFunction) OR (40C) ;  *type 1 transfer
bbFUN11:        T ← (304C) , AT[BBF,11] ;                 *salufop ← [1,0,r or t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (100C) ; *type 2 transfer
bbFUN12:        T ← (363C) , AT[BBF,12] ;                 *salufop ← [1,0,r xor t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (100C) ; *type 2 transfer
bbFUN13:        T ← (327C) , AT[BBF,13] ;                 *salufop ← [1,0,r and not t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (100C) ; *type 2 transfer
bbFUN14:        T ← (204C) , AT[BBF,14] ;                 *salufop ← [0,0,r or t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (200C) ; *type 4 transfer
bbFUN15:        T ← (304C) , AT[BBF,15] ;                 *salufop ← [1,0,r or t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (140C) ; *type 3 transfer
bbFUN16:        T ← (363C) , AT[BBF,16] ;                 *salufop ← [1,0,r xor t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (140C) ; *type 3 transfer
bbFUN17:        T ← (327C) , AT[BBF,17] ;                 *salufop ← [1,0,r and not t]
        GOTO[bbEndInit] , bbFunction ← (bbFunction) OR (140C) ; *type 3 transfer

bbEndInit:
        LOADPAGE[BBP1] ;
        GOTOP[bbFirstItem],saluf ← t;
```

```
        ONPAGE[BBP1] ;
*       Inner loops
*               functions 0-7
*
bbInnerLoops:
bbILA1Z:
        call[.+1] , AT[BBIDISP,bbILtype0] ;
        DBLGOTO[bbILA2,bbILAX,MB] ,
        T ← (BBFA[SB[bbSOURCE]]) or (t) ;
bbILA2: DISP[bbILA1] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) ;
bbILAX: DISP[bbILA1] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) ;
bbILA1: return , AT[BBILA,BBNRM] ;
%       bbila2 will go to
*               bbila1          no refill required
*               bbilas          source refill required
*               bbilad          dest refill required
*               bbilasd         source and dest refills required
*               bbilai          item refill required
*
%
*               function 10
*
bbILB1:
                call[.+1] , AT[BBIDISP,bbILtype1] ;
                T ← (BBFA[SB[bbSOURCE]]) or (t) ;
bbILB2: DISP[bbILB3] , RTEMP ← T ;      *5 copies of the following code are
                                *required to save the dispatch data
                                *from the BBFA executed previously
%       bbilb2 will go to
*               bbilb3          no refill required
*               bbilb3s         source refill required
*               bbilb3d         dest refill required
*               bbilb3sd        source and dest refills required
*               bbilb3i         item refill required
%
bbILB3: DB[bbDEST] ← (DB[bbDEST]) AND NOT(T) , AT[BBILB,BBNRM] ;
        T ← RTEMP ;
        T ← (bbGRY) AND (T) ;
bbILB6: return , DB[bbDEST] ← (BBFB[DB[bbDEST]]) OR (T) ;
*
bbILB3S:        DB[bbDEST] ← (DB[bbDEST]) AND NOT(T) , AT[BBILB,BBSRC] ;
        T ← RTEMP ;
        T ← (bbGRY) AND (T) ;
bbILB6S:        GOTO[bbILBS] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) OR (T) ;
*
bbILB3D:        DB[bbDEST] ← (DB[bbDEST]) AND NOT(T) , AT[BBILB,BBDST] ;
        call[bbtgry],T ← RTEMP ;
bbILB6D:        GOTO[bbILBD] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) OR (T) ;
*
bbILB3SD:       DB[bbDEST] ← (DB[bbDEST]) AND NOT(T) , AT[BBILB,BBBTH] ;
        call[bbtgry],T ← RTEMP ;
bbILB6SD:       GOTO[bbILBSD] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) OR (T) ;
*
bbILB3I:        DB[bbDEST] ← (DB[bbDEST]) AND NOT(T) , AT[BBILB,BBITM] ;
        call[bbtgry],T ← RTEMP ;
bbILB6I:        GOTO[bbILBI] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) OR (T) ;
bbtgry:
        T ← (bbGRY) AND (T),return ;
```

```
*               functions 11-13

bbILC1:
                call[.+1] , AT[BBIDISP,bbILtype2] ;
                T ← (BBFA[SB[bbSOURCE]]) or (t) ;
bbILC2:         DISP[bbILC3] , T ← (bbGRY) AND (T) ;
bbILC3:         return , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILC,BBNRM] ;
%       bbilc2 will go to
*               bbilc3          no refill required
*               bbilc3s         source refill required
*               bbilc3d         dest refill required
*               bbilc3sd        source and dest refills required
*               bbilc3i         item refill required
%
*               functions 14

bbILE1x:
                call[.+1] , AT[BBIDISP,bbILtype4] ;
                T ← (BBFA[AllOnes]) ;
bbILE3: DISP[bbILE4] , T ← (bbGRY) and (t) ;
bbILE4: return , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) , AT[BBILE,BBNRM] ;
%       bbile3 will go to
*               bbile4          no refill required
*               bbile4s         source refill required
*               bbile4d         dest refill required
*               bbile4sd                source and dest refills required
*               bbile4i         item refill required
%


*               functions 15-17

bbILD1:
                call[.+1] , AT[BBIDISP,bbILtype3] ;
                T ← BBFA[AllOnes] ;
bbILD3:         DISP[bbILD4] , T ← (bbGRY) AND (T) ;
bbILD4:         return , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILD,BBNRM] ;
%       bbild3 will go to
*               bbild4          no refill required
*               bbild4s         source refill required
*               bbild4d         dest refill required
*               bbild4sd        source and dest refills required
*               bbild4i         item refill required
%
```

```
*Source refill
bbSourceRefill:
bbILBS:
bbILAS: DBLGOTO[bbNoSrcFetch,bbSrcFetch,R ODD] , bbSrcQAddrLo ← (bbSrcQAddrLo) + (4C) , AT[BBILA,BBSRC] ;
bbILC3S:        GOTO[bbSourceRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILC,BBSRC] ;
bbILD4S:        GOTO[bbSourceRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILD,BBSRC] ;
bbILE4S:        GOTO[bbSourceRefill] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) , AT[BBILE,BBSRC] ;

bbILret:
bbNoSrcFetch:
        return ;
bbSrcFetch:
        goto[.+3,nocarry];
        bbSrcQAddrHi←(bbSrcQAddrHi)+(400c)+1;
        nop;*can't load hi base reg in m-i preceding a memory operation
        PFETCH4[bbSrcQAddrLo,bbSOURCE,0] , return ;

*Dest refill
bbDestRefill:
        nop;
bbDestRefillx:
        GOTO[bbDestFetch] , PSTORE4[bbDestQAddrLo,bbDEST,0];
bbILBD:
bbILAD: GOTO[bbDestRefillx] , AT[BBILA,BBDST] ;
bbILC3D:        GOTO[bbDestRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILC,BBDST] ;
bbILD4D:        GOTO[bbDestRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILD,BBDST] ;
bbILE4D:        GOTO[bbDestRefill] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) , AT[BBILE,BBDST] ;

bbDestFetch:
        bbDestQAddrLo ← (bbDestQAddrLo) + (4C);
        goto[.+3,nocarry];
        bbDestQAddrHi←(bbDestQAddrHi)+(400c)+1;
        nop;*can't load hi base reg in m-i preceding a memory operation
        goto[bbildisp] , PFETCH4[bbDestQAddrLo,bbDEST,0] ;
```

```
*	Source and Dest refill
bbSrcDestRefill:
bbILBSD:
bbILASD:	DBLGOTO[bbDestRefill,bbSrcDestFetch,R ODD] , bbSrcQAddrLo ← (bbSrcQAddrLo) + (4C) , AT[BBILA,BBBTH] ;
bbILC3SD:	GOTO[bbSrcDestRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILC,BBBTH] ;
bbILD4SD:	GOTO[bbSrcDestRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILD,BBBTH] ;
bbILE4SD:	GOTO[bbSrcDestRefill] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) , AT[BBILE,BBBTH] ;

bbSrcDestFetch:
	goto[.+3,nocarry];
	bbSrcQAddrHi←(bbSrcQAddrHi)+(400c)+1;
	nop;*can't load hi base reg in m-1 preceding a memory operation
	PFETCH4[bbSrcQAddrLo,bbSOURCE,0] , goto[bbDestRefill] ;
```

```
*	Common return to inner loops
bbILDISP:
	call[bbILret] , BBFB ;
	DISPATCH[bbFunction,10,4] ;
	DISP[bbInnerLoops] ;
%			depending upon the loop type this will go to
*		type 0	bbila1z		function 0-7
*		type 1	bbilb1		function 10
*		type 2	bbilc1		function 11-13
*		type 3	bbild1		function 15-17
*		type 4	bbile1x		function 14
%
```

```
*       Item refill
*                       test if right to left or left to right
bbItemRefill:
bbILBI:
bbILAI: GOTO[bbILI] , lu←ldf[bbFunction,7,1] , AT[BBILA,BBITM] ;
bbILC3I:        GOTO[bbItemRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILC,BBITM] ;
bbILD4I:        GOTO[bbItemRefill] , DB[bbDEST] ← (BBFBX[DB[bbDEST]]) SALUFOP (T) , AT[BBILD,BBITM] ;
bbILE4I:        GOTO[bbItemRefill] , DB[bbDEST] ← (BBFB[DB[bbDEST]]) SALUFOP (T) , AT[BBILE,BBITM] ;

bbILI:  goto[bbContRtoL,alu#0] ;
```

```
*               common item refill
bbComItemRefill:
        pstore4[bbDestQAddrLo,bbDEST,0] ;
        call[bbILret],stack ← (stack) + 1 ;*update icom (TOS if called by Mesa, AC1 if called by Nova)
        GOTO[.+3,R >= 0] , bbItemsRemainingMinus2 ← (bbItemsRemainingMinus2) - 1 ;
bbExitp1:       LoadPage[bbp2];
        lu ← ldf[bbFunction,1,1], gotop[bbExit]; *Completion return
*Test for interrupts
        lu ← NWW,skip[R>=0];
        dblgoto[bbDestUpdate,bbSrcUpdate,R ODD],lu←bbSrcQAddrLo; *Interrupts disabled by Nova
        skip[ALU#0],lu ← ldf[bbFunction,1,1];
        dblgoto[bbDestUpdate,bbSrcUpdate,R ODD],lu←bbSrcQAddrLo;
*//******* Start of Alto Code ********************************************
        dblgoto[bbMesaInt,bbNovaInt,ALU#0];
*//******* End of Alto Code **********************************************

bbMesaInt:      LoadPage[opPage3];
        T ← 1c, callp[MIPend];  *back up the Mesa PC by 1
        dblgoto[bbDestUpdate,bbSrcUpdate,R ODD],lu←bbSrcQAddrLo;

*Since we are now in a subroutine, and since the Nova only
*checks for interrupts at buffer refill and during jumps,
*things are complicated.  We simulate a JMP .+0

*//******* Start of Alto Code ********************************************
bbNovaInt:      T ← ldf[GETRSPEC[127],15,2];
         LoadPage[nePage]; *Point the jump-to PC at the BitBLT
        T ← (PCB) + (T), gotop[JMP];
*//******* End of Alto Code **********************************************


bbSrcUpdate:
        skip[r>=0],t ← lsh[bbSBMR,4] ;
        bbSrcStartBitHi←(bbSrcStartBitHi)-(20c);
        bbSrcStartBitLo ← (bbSrcStartBitLo) + (t) ;
        skip[nocarry],t←rsh[bbSBMR,14];
        bbSrcStartBitHi ← (bbSrcStartBitHi) + 1 ;
        bbSrcStartBitHi ← (bbSrcStartBitHi) + (t);

bbDestUpdate:
        skip[r>=0],t ← lsh[bbDBMR,4] ;
        bbDestStartBitHi←(bbDestStartBitHi)-(20c);
        bbDestStartBitLo ← (bbDestStartBitLo) + (t) ;
        skip[nocarry],t←rsh[bbDBMR,14];
        bbDestStartBitHi ← (bbDestStartBitHi) + 1 ;
        bbDestStartBitHi ← (bbDestStartBitHi) + (t);

bbFirstItem:
bbTouchSourcePages:
        call[bbILret];*task switch
        GOTO[bbTouchDestPages,R ODD],lu←bbSrcQAddrLo ;
        t←(bbMinusItemWidth)+1;
        RTEMP←(zero)-(t);
        call[bbSetbbSQA];
        t←(bbSrcStartBitLo) and not (170000c);
        RTEMP←(RTEMP)+(t);
        skip[carry],RTEMP←rsh[RTEMP,14];
        goto[bbTSP1];
        skip[alu#0];
        RTEMP←(20c);

bbTSP1:
        t←RTEMP←lsh[RTEMP,10];
        call[.+1];*avoid mem pass-around
        pfetch4[bbSrcQAddrLo,bbSOURCE];
        t←RTEMP←(RTEMP)-(400c);
        skip[alu<0];
        return;
        nop;

bbTouchDestPages:
        t←(bbMinusItemWidth)+1;
        RTEMP←(zero)-(t);
        call[bbSetbbDQA];
        t←(bbDestStartBitLo) and not (170000c);
        RTEMP←(RTEMP)+(t);
        skip[carry],RTEMP←rsh[RTEMP,14];
        goto[bbTDP1];
        skip[alu#0];
        RTEMP←(20c);

bbTDP1:
        t←RTEMP←lsh[RTEMP,10];
        call[.+1];*avoid mem pass-around
        pfetch4[bbDestQAddrLo,bbDEST];
        t←RTEMP←(RTEMP)-(400c);
        skip[alu<0];
        return;

bbNewGry:
        lu←ldf[bbFunction,14,1];
        goto[bbNoGray,alu=0],MNBR ← bbMinusItemWidth ;
        t←(AC2)+(14c) ;
        lu ← ldf[bbFunction,1,1]; *"called from Mesa" bit
        dblgoto[NovaGray,MesaGray,ALU=0], t←(ldf[bbGrayCnt,16,2])+(t);
NovaGray:       pfetch1[Nova,bbgry], goto[.+2];
MesaGray:       Pfetch1[MDS,bbgry];
        bbGrayCnt←(bbGrayCnt)+1;
bbNoGray:
        SB←bbSrcStartBitLo;
```

```
skip[r even],lu←bbSrcQAddrLo ;
SB ← bbDestStartBitLo ;
DB←bbDestStartBitLo;
lu←ldf[bbFunction,7,1] ;
skip[alu#0] ;
GOTO[bbILDISP] ;
```

```
bbNewRtoL:      *new item right to left
        t←bbMinusSDNonOverlap ;
        task,bbMinusNumBitsTran←t ;
        t←MNBR←bbMinusNumBitsTran ;
        t←(bbMinusItemWidth)-(t) ;
        call[bbILret],bbMinusBitsRemaining ←t;
bbSourceBitUpdate:
        goto[bbDestBitUpdate,r odd],lu←bbSrcQAddrLo;
        bbSrcStartBitLo ← (bbSrcStartBitLo) - (t) ;
        skip[nocarry] ;
        bbSrcStartBitHi ← (bbSrcStartBitHi) + 1 ;
        RTEMP←t;
        call[bbSetbbSQA];
        t←RTEMP;
        SB←bbSrcStartBitLo;
bbDestBitUpdate:
        bbDestStartBitLo ← (bbDestStartBitLo) - (t) ;
        skip[nocarry] ;
        bbDestStartBitHi ← (bbDestStartBitHi) + 1 ;
        goto[bbGetNextSubItem];

bbContRtoL:     *continue item right to left
        t ← bbMinusBitsRemaining ;
        skip[alu#0] , lu ← (bbMinusSDNonOverlap) - (t) ;
        GOTO[bbComItemRefill] ;*current item exhausted
        skip[nocarry],PSTORE4[bbDestQAddrLo,bbDEST,0];
        t ← bbMinusSDNonOverlap ;
        bbMinusNumBitsTran ← t ;
        task,bbMinusBitsRemaining ← (bbMinusBitsRemaining) - (t) ;
        mnbr ← bbMinusNumBitsTran ;
*         adjust source and dest bit addresses for next sub-item
        goto[bbrlskip,r odd],lu←bbSrcQAddrLo;
        t ← bbMinusNumBitsTran ;
        bbSrcStartBitLo ← (bbSrcStartBitLo) + (t) ;
        skip[carry] ;
        bbSrcStartBitHi ← (bbSrcStartBitHi) - 1 ;
        call[bbSetbbSQA];
        SB←bbSrcStartBitLo;
bbrlskip:
        t ← bbMinusNumBitsTran ;
        bbDestStartBitLo ← (bbDestStartBitLo) + (t) ;
        skip[carry] ;
         bbDestStartBitHi ← (bbDestStartBitHi) - 1 ;
bbGetNextSubItem:
        call[bbSetbbDQA];
        skip[r even],lu←bbSrcQAddrLo ;
        SB ← bbDestStartBitLo ;
        DB←bbDestStartBitLo;
        pfetch4[bbDestQAddrLo,bbDEST,0] ;
        skip[R ODD] , lu ← bbSrcQAddrLo ;
        pfetch4[bbSrcQAddrLo,bbSOURCE,0] ;
        GOTO[bbILDISP] ;
```

```
bbSetbbSQA:     *build source mem-base from source bit
        t←rsh[bbSrcStartBitLo,4] ;
        bbSrcQAddrLo ← t ;
        t←lsh[bbSrcStartBitHi,14] ;
        bbSrcQAddrLo ← (bbSrcQAddrLo) + (t) ;
        t←rsh[bbSrcStartBitHi,4] ;
        bbSrcQAddrHi ← t ;
        bbSrcQAddrHi←(lsh[bbSrcQAddrHi,10]) + (t) + 1 ;
        return,bbSrcQAddrLo ← (bbSrcQAddrLo) and not (3c) ;

bbSetbbDQA:     *build dest mem-base from dest bit
        t←rsh[bbDestStartBitLo,4] ;
        bbDestQAddrLo ← t ;
        t←lsh[bbDestStartBitHi,14] ;
        bbDestQAddrLo ← (bbDestQAddrLo) + (t) ;
        t←rsh[bbDestStartBitHi,4] ;
        bbDestQAddrHi ← t ;
        bbDestQAddrHi←(lsh[bbDestQAddrHi,10]) + (t) + 1 ;
        return,bbDestQAddrLo ← (bbDestQAddrLo) and not (3c) ;

        end[bb];
```

```
* MicroD 8.11 (OS 16) of July 1, 1979
*   at 17-Oct-79 11:57:40

INSERT[OccupiedDofs];

        TITLE[BMesa1Occupied];

* Locations reserved on page 0

        IMRESERVE[0, 100, 233];
        IMRESERVE[0, 335, 23];
        IMRESERVE[0, 361, 17];

* Locations reserved on page 1

        IMRESERVE[1, 100, 253];

* Locations reserved on page 2

        IMRESERVE[2, 100, 225];
        IMRESERVE[2, 340, 1];
        IMRESERVE[2, 372, 1];
        IMRESERVE[2, 374, 1];
        IMRESERVE[2, 376, 1];

* Locations reserved on page 3

        IMRESERVE[3, 0, 336];
        IMRESERVE[3, 370, 3];
        IMRESERVE[3, 374, 3];

* Locations reserved on page 7

        IMRESERVE[7, 27, 1];
        IMRESERVE[7, 76, 1];

* Locations reserved on page 10B

        IMRESERVE[10, 0, 363];

* Locations reserved on page 12B

        IMRESERVE[12, 0, 375];

* Locations reserved on page 13B

        IMRESERVE[13, 0, 305];

* Locations reserved on page 14B

        IMRESERVE[14, 0, 137];

* Locations reserved on page 15B

        IMRESERVE[15, 300, 1];

* Locations reserved on page 16B

        IMRESERVE[16, 0, 312];

* Locations reserved on page 17B

        IMRESERVE[17, 140, 40];

        END;
```

```
* D0Lang.mc
        *Last edited:  July 2, 1979  5:10 PM by Chang, add IMUNRESERVE, GoExt
        * edited:  June 27, 1979  10:52 AM by Johnsson
        * edited:  April 23, 1979  3:05 PM by Jarvis
        * edited:  December 16, 1978  2:16 PM by CPT
BUILTIN[M@,2];  *Declare macro
BUILTIN[N@,3];  *Declare neutral
BUILTIN[MEMORY@,4];      *Declare Memory[name,wordsize,length,srcmacro,
                *sinkmacro,tagmacro,postmacro]
BUILTIN[TARGET@,5];
BUILTIN[DEFAULT@,6];
BUILTIN[FLX@,7];         *Declare field
BUILTIN[PF@,10];         *Preassign value to field
BUILTIN[SET,11];         *Declare integer and set value
BUILTIN[ADD,12];         *Add up to 8 integers
BUILTIN[IP,13]; *Integer part of address
BUILTIN[IFSE@,14];       *If string equal (IFSE@[s1,s2,true,false])
BUILTIN[IFA@,15];        *If field assigned (IFA@[field,true,false])
BUILTIN[IFE@,16];        *If integers equal
BUILTIN[IFG@,17];        *If integer 1 > integer 2
BUILTIN[IDF@,20];        *If symbol in symbol table and not unbound address
*BUILTIN[IFME@,21];      *If memory part of address equals string
BUILTIN[ER@,22];         *Error message (ER@[string,abortflag,integer])
BUILTIN[LIST@,23];       *Set listing mode for memory
BUILTIN[INSERT@,24];     *Insert file
BUILTIN[NOT@,25];        *1'S complement
BUILTIN[REPEAT@,26];     *Repeat the text #2 #1 times
BUILTIN[OR@,27];         *Inclusive or up to 10 integers
BUILTIN[XOR@,30];        *Exclusive or up to 10 integers
BUILTIN[AND@,31];        *And up to 10 integers
BUILTIN[COMCHAR@,32];    *Set comment char for conditional assemblies
*BUILTIN[BITTABLE@,33]; *Makes #1 a bit table of length #2 bits
*BUILTIN[GETBIT@,34];    *Is the bit in bittable #1 at pos. #2
*BUILTIN[SETBIT@,35];    *SETBIT[table,1stbit,nbits,distance,value]
*BUILTIN[FINDBIT@,36];   *FINDBIT[TABLE@,1STBIT,NBITS,DISTANCE,HOPDISTANCE,NHOPS]
*BUILTIN[MEMBT@,37];     *MEMBT[memory,table] creates a bit table for memory
BUILTIN[LSHIFT,40];      *Shifts the integer #1 left #2 positions
BUILTIN[RSHIFT,41];      *Shifts the integer #1 right #2 positions
BUILTIN[FVAL@,42];       *FVAL@[field] is an integer whose value is the
                *current contents of the field
BUILTIN[SELECT@,43];     *#1 is an integer .ge. 0 and .le. 7. evaluates
                *#2 if #1 = 0, ..., #9 if #1 = 7.  Error if #1 > 7
BUILTIN[SETPOST@,44];    *Set post-evaluation macro (SETPOST@[mem,macro])
BUILTIN[SETMBEXT@,47];   *Set .mb file extension

COMCHAR@[~];             *Makes "*~" work like "%" at beginning of lines

SETMBEXT@[DIB];

M@[SUB,ADD[#1,NOT@[#2],1]];

%Memory declarations must have names and sizes agreeing with those in
Midas, except that IM must agree with the form expected by MicroD.
%
MEMORY@[IM,124,10000,W@,W@];
MEMORY@[RM,20,400,RSRC@,RSINK@];
MEMORY@[IMLOCK,1,10000,W@,W@];
MEMORY@[VERSION,20,1,W@,W@];

M@[W@,];                        *Dummy macro required for memory definitions
IM[ILC@,0];              *Location counter for IM

%Memory lock Control macro
IMRESERVE[page #,first address,number of addresses]
 Will not allocate in the reserved locations
%
FLX@[LOCK@,0,0];

M@[IMRESERVE,IMLOCK[Z@,ADD[LSHIFT[#1,10],#2]]
   REPEAT@[#3,Z@[(LOCK@[1])]]
        ]
VERSION[VLC@,0];

M@[IMUNRESERVE,IMLOCK[Z@,ADD[LSHIFT[#1,10],#2]]
   REPEAT@[#3,Z@[(LOCK@[0])]]
        ]
VERSION[VLC@,0];

FLX@[VERS@,0,17];

%Second arg of LIST controls listing of memories as follows:
1 = (TAG) nnnn nnnn nnnn ...
2 = (TAG) F1←3, F2←4, ...
4 = numerically-ordered list of address symbols
10 = alphabetically-ordered list of address symbols
%
LIST@[IM,7]; LIST@[RM,5]; LIST@[,17];
```

```
%Three macros define parameters from which constants, RM values, or
IM data can be constructed:
   MP[NAME,octalstring] makes a parameter of NAME;
   SP[NAME,P1,P2,P3,P4,P5,P6,P7,P8] makes a parameter NAME equal to the sum
of P1, P2, P3, P4, P5, P6, P7, and P8, where the Pn may be parameters or
addresses.
   NSP[NAME,P1,P2,P3,P4,P5,P6,P7,P8] is ones complement of SP.

The parameter "NAME" is defined by the integer "NAME!", so it is ok to
use "NAME" for a constant as well as a paramtor.  However, it is illegal
to define constants, addresses, etc. with identical names.

"Literal" constants such as "322C", "177622C", or "32400C" may be
inserted in microinstructions without previous definition.

Alternatively, constants may be constructed from parameters, integers, and
addresses using the following macros:
   MC[NAME,P1,P2,P3,P4,P5,P6,P7,P8] defines name as a constant with value =
sum of parameters P1, P2, P3, P4, P5, P6, P7, and P8;
   NMC[NAME,P1,P2,P3,P4,P5,P6,P7,P8] is the ones complement of MC.

Note:  MC and NMC also define NAME as a parameter.
%

*Fields for initializing 16-bit wide memories
FLX@[E0@,0,3]; FLX@[E1@,4,17];

*Macro to initialize (16-bit) variables in the target memory.  This is
*done by writing 32100V (i.e., as a literal).
M@[V,E1@[#1] E0@[#2]];

M@[MP@,SET[#1!,#2]];
M@[SP@,IFG@[#0,11,ER@[Too.many.args.for.#1],
  SET[#1!,DPS@[#2,#3,#4,#5,#6,#7,#8,#9]]]];
M@[NSP@,IFG@[#0,11,ER@[Too.many.args.for.#1],
  SET[#1!,NOT@[DPS@[#2,#3,#4,#5,#6,#7,#8,#9]]]]];
M@[DPS@,ADD[PX@[#1],PX@[#2],PX@[#3],PX@[#4],PX@[#5],PX@[#6],PX@[#7],PX@[#8]]];
M@[!,0];
M@[PX@,IDF@[#1!,#1!,#1]];


M@[C,IFA@[F1@,ER@[F1.used.twice],
       IFA@[F2@,ER@[F2.used.twice],
        IFE@[AND@[#3#2,177760],0,
                   SET[!T1@,ADD[LSHIFT[#2,4],RSHIFT[#1,10]]]
                   SET[!T2@,AND@[#1,377]]
          IFE@[!T2@,0,BS@[1] FF@[!T1@],
            BS@[0] FF@[!T2@]
              ] B,ER@[Constant.too.big]
            ]
           ]
          ]
  ];

M@[MC,IFG@[#0,11,ER@[Too.many.args.for.#1],
  SP@[#1,DPS@[#2,#3,#4,#5,#6,#7,#8,#9]]
  M@[#1,ADD[#1!]C]]];

M@[NMC,IFG@[#0,11,ER@[Too.many.args.for.#1],
  SP@[#1,NOT@[DPS@[#2,#3,#4,#5,#6,#7,#8,#9]]]
  M@[#1,ADD[#1!]C]]];
```

```
%RM stuff

RM constants and variables are allocated in two steps.  First, the
group of 100 registers that must contain the ones being allocated is
declared by SETTASK, which binds the integer RMBASE to the top two
bits of the selected task times 100.

Then registers in that group of 100 are allocated as follows:
        RV[FOO,23,p1,...,p7];    *Creates FOO = RM 23, value sum of params
        RV[FOO,23];              *Creates FOO = RM 23, no value
        RV[FOO];                 *Creates FOO at last location + 1
        RV[FOO,,17575];          *Creates address FOO at location after
                                 *last one allocated with value 17575
These macros leave the integer RLi, where i = 0 to 3 (the region)
bound to the last displacement allocated, and the integer RL is always
equal to RLi for the current region.
%

M@[SETTASK,IFG@[#1,17,ER@[Illegal.SETTASK],
  SELECT@[QTASK@,SET[RL0@,RL@],SET[RL1@,RL@],SET[RL2@,RL@],SET[RL3@,RL@]]
  SET[QTASK@,RSHIFT[#1,2]] SET[CURTSK@,#1]
  SET[RMBASE@,LSHIFT[QTASK@,6]]          *For reg. instructions
  SET[BRBASE@,LSHIFT[CURTSK@,4]]                 *For mem. ref instructions
  SET[RL@,SELECT@[QTASK@,RL0@,RL1@,RL2@,RL3@]]
]];

SET[RL0@,177777]; SET[RL1@,177777];
SET[RL2@,177777]; SET[RL3@,177777];
SET[RMBASE@,0]; SET[QTASK@,0]; SET[CURTSK@,0]; SET[RL@,177777];

M@[RV,SET[RL@,IFSE@[#2,,ADD[1,RL@],#2]]
  IFG@[RL@,77,ER@[RM.ovf]]
  RM[#1,ADD[RMBASE@,RL@]] RM[RLC@,IP[#1]]
  IFG@[#0,2,RLC@[DPS@[#3,#4,#5,#6,#7,#8,#9]V]]
];

*RM addresses used as sources/destinations execute following macros.
M@[RSINK@,CKRMA@[#1] RB←];
M@[RSRC@,CKRMA@[#1] RB];

M@[CKRMA@, SET[ZOT@,IP[#1]] MRS@[AND@[ZOT@,77]]
  IFE@[RSHIFT[ZOT@,6],QTASK@,
    IFE@[AND@[ZOT@,60],0,
      IFE@[AND@[CURTSK@,3],0,,ER@[#1.unaddressable]]
    ],ER@[#1.unaddressable]]
];

*PCF[RMADDR], SB[RMADDR], and DB[RMADDR] are also A sources
M@[PCF,QRS@[#1,0,PCF]];
M@[SB,QRS@[#1,1,SB]];
M@[DB,QRS@[#1,2,DB]];

M@[QRS@,SET[ZOT@,IP[#1]]
  IFE@[AND@[ZOT@,3],0,IFE@[RSHIFT[ZOT@,6],QTASK@,MRS@[ADD[100,ZOT@,#2]]RB,
      ER@[#1.in.illegal.RM.region]],ER@[#1.not.quadaligned]]];

*other register sources
M@[ALURESULT,IFA@[MRS1X@,MRS1ER@,
  MRS@[107] LDF[RB,4,4]]];        *aluresult = ovf,car,=0,<0
M@[SALUF,IFA@[MRS1X@,MRS1ER@,
  MRS@[107] LDF[RB,10,10]]];
M@[SSTKP,IFA@[MRS1X@,MRS1ER@,
  MRS@[103] LDF[RB,0,10]]];
M@[NSTKP,IFA@[MRS1X@,MRS1ER@,
  MRS@[103] LDF[RB,10,10]]];
M@[STKP,IFA@[MRS1X@,MRS1ER@,
  MRS@[103] LDF[RB,10,10]]];
M@[MEMERROR,IFA@[MRS1X@,MRS1ER@,
  MRS@[117] RB]];
M@[MEMSYNDROME,IFA@[MRS1X@,MRS1ER@,
  MRS@[113] RB]];
M@[DBXREG,IFA@[MRS1X@,MRS1ER@,
  MRS@[127] LDF[RB,0,4]]];
M@[MWXREG,IFA@[MRS1X@,MRS1ER@,
  MRS@[127] LDF[RB,4,4]]];
M@[CYCLECONTROL,IFA@[MRS1X@,MRS1ER@,
  MRS@[127] LDF[RB,0,10]]];
M@[PCXREG,IFA@[MRS1X@,MRS1ER@,
  MRS@[127] LDF[RB,10,4]]];
M@[PCFREG,IFA@[MRS1X@,MRS1ER@,
  MRS@[127] LDF[RB,14,4]]];
M@[PRINTER,IFA@[MRS1X@,MRS1ER@,
  REGSHIFT[] MRS@[127] RB]];
M@[DBSB,IFA@[MRS1X@,MRS1ER@,
  REGSHIFT[] MRS@[133] RB]];
M@[TIMER,IFA@[MRS1X@,MRS1ER@,
  MRS@[133] RB]];
M@[RS232,IFA@[MRS1X@,MRS1ER@,
  MRS@[137] RB]];
M@[MNBR,IFA@[MRS1X@,MRS1ER@,
  REGSHIFT[] MRS@[137] RB]];
M@[APCTASK,IFA@[MRS1X@,MRS1ER@,
  MRS@[143] LDF[RB,0,4]]];
M@[APC,IFA@[MRS1X@,MRS1ER@,
  MRS@[143] LZERO[RB,4]]];
M@[APCTASK&APC,IFA@[MRS1X@,MRS1ER@,
  MRS@[143] RB]];
M@[APC&APCTASK,IFA@[MRS1X@,MRS1ER@,
  MRS@[143] RB]];
M@[CTASK,IFA@[MRS1X@,MRS1ER@,
```

```
  MRS@[147] LDF[RB,0,4]]];
M@[NCIA,IFA@[MRS1X@,MRS1ER@,
  MRS@[147] LZERO[RB,4]]];
M@[CSDATA,IFA@[MRS1X@,MRS1ER@,
  MRS@[153] RB]];
M@[PAGE,IFA@[MRS1X@,MRS1ER@,
  MRS@[157] LDF[RB,0,4]]];
M@[PARITY,IFA@[MRS1X@,MRS1ER@,
  MRS@[157] LDF[RB,4,4]]];
M@[BOOTREASON,IFA@[MRS1X@,MRS1ER@,
  MRS@[157] LDF[RB,10,10]]];
M@[MRS1ER@,ER@[MRS1.used.twice]];

*To get a multi-field word using only one specification , use
*GETRSPEC[mrs address] .  Thus to load T with APCTASK and APC, use *the following syntax:
*                  T ← GETRSPEC[143];

M@[GETRSPEC,IFA@[MRS1X@,ER@[MRS1.used.twice],
  MRS@[#1] RB]];

*StkP sources/destinations and other RB sources are defined with the
*following macros:
*MKRSRC@[name,rselvalue] defines the source -or-
*MKRDEST@[name,rselvalue] defines the destination
M@[MKRSRC@,M@[#1,(MRS@[#2] #3) RB]];
M@[MKRDEST@,M@[#1,(MRS@[#2] #3) RB←]];

*STACKSHIFT (StackShift) is F2
MKRSRC@[STACK,163,]; MKRSRC@[STACK&+1,167,];
MKRSRC@[STACK&-1,173,]; MKRSRC@[STACK&-2,177,];
MKRSRC@[STACK&+2,163,STACKSHIFT]; MKRSRC@[STACK&+3,167,STACKSHIFT];
MKRSRC@[STACK&-3,177,STACKSHIFT];
MKRDEST@[STACK←,163,]; MKRDEST@[STACK&+1←,167,];
MKRDEST@[STACK&-1←,173,]; MKRDEST@[STACK&-2←,177,];
MKRDEST@[STACK&+2←,163,STACKSHIFT]; MKRDEST@[STACK&+3←,167,STACKSHIFT];
MKRDEST@[STACK&-3←,177,STACKSHIFT];
```

```
%IM used as data stuff

IM words can be assembled as data using the "LH" (left-half) and "RH"
(right-half) macros defined below. Each of these takes up to 8 arguments
which are either parameters, addresses, or integers. These are summed
to form the value stored.

The way to assemble data is:
        DATA[(LH[...] RH[...] AT[...])];
%

FLX@[V0@,0,17]; FLX@[V1@,20,37];

M@[LH,IFG@[#0,10,ER@[Too.many.args.for.LH]]
  V0@[DPS@[#1,#2,#3,#4,#5,#6,#7,#8]]];

M@[RH,IFG@[#0,10,ER@[Too.many.args.for.RH]]
  V1@[DPS@[#1,#2,#3,#4,#5,#6,#7,#8]]];

M@[DATA,ILC@[RETCL@[2] #1]];     *Indicate "Return" so no MicroD fixup


*IM field definitions
M@[MRS@,MRS1@[RSHIFT[#1,2]] RSEL2@[AND@[#1,3]]];         *MEMINS@, RMOD@, and RSEL@
M@[MRS1@,MRS1X@[XOR@[#1,14]]];
FLX@[MRS1X@,0,5];
M@[RSMOD@,RSMOD1@[RSHIFT[#1,2]] RSEL2@[AND@[#1,3]]];     *RMOD@ and RSEL@
M@[RSMOD1@,RSMOD1X@[XOR@[#1,14]]];
FLX@[RSMOD1X@,1,5];
FLX@[F2@,22,25];                 *FF[4:7]
FLX@[JC@,26,30];                 *Jump control
M@[JA@,JA1@[RSHIFT[#1,6]] JA2@[AND@[#1,77]]];
FLX@[JA1@,122,123];
FLX@[JA2@,31,36];
FLX@[JA7@,36,36];
FLX@[RSEL2@,120,121];

*Fields in regular instructions
FLX@[AF0@,6,6]; *Sign bit for constants
FLX@[ALF@,6,11];                 *Entire ALUF field
FLX@[BS@,12,13];                 *Source for BMux
FLX@[LR@,20,20];                 *Load RM
FLX@[LT@,21,21];                 *Load T
FLX@[IMPARITY@,37,37];   *Parity bit
M@[FF@,F1@[RSHIFT[#1,4]] F2@[AND@[#1,17]]];      *Function field
FLX@[F1@,14,17];                 *FF[0:3]

*Fields in memory reference instructions
FLX@[TYPE@,6,11];         *Type of memory reference
*Source/destination
FLX@[SRCDES@,12,21];
*Extra stuff for MicroD
FLX@[BRKP@,40,40];        *Instruction has a breakpoint
FLX@[W0@,44,57];
*FLX@[@GW0@,41,57];       *@W0, GLB, and PW0
FLX@[@W0@,41,41];         *Place at absolute loc. W0
*FLX@[GLB@,42,42];        *Place at global call loc.
FLX@[PW0@,43,43];         *Bit 43=1 tells MicroD that 44:47 contain the page
                *on which the instruction should be placed
FLX@[PGE@,44,47];         *contains page for mi placement

FLX@[RETCL@,60,61];       *2 = does a Return
                *1 = does a Call
FLX@[ODDCALL@,62,62];*Does a call from an odd location
FLX@[SWPAGE@,63,63];      *This instruction loads the PAGE register
FLX@[W1@,64,77];          *Imaginary address of unconditional or false
                *branch address
FLX@[CHPAGE@,101,101];*Indicates F-field is LOADPAGE, new page in F2
*FLX@[EMUL@,102,102];*Presently unused
FLX@[CND@,103,103];       *Has a branch condition
FLX@[W2@,104,117];        *Imaginary address of conditional branch when true


M@[FZ@,IFA@[F1@,ER@[F1.used.twice],IFA@[F2@,ER@[F2.used.twice],FF@[#1] ]]];
M@[FF1@,IFA@[F1@,ER@[F1.used.twice],F1@[#1] ]];
M@[FF2@,IFA@[F2@,ER@[F2.used.twice],F2@[#1] ]];

M@[BREAKPOINT,BRKP@[1]?];
```

```
*Neutrals and connection macros
N@[RB←]; N@[RB]; N@[A]; N@[B←]; N@[B]; N@[?]; N@[T←]; N@[T]; N@[LU]; N@[LU←];

M@[A←A,A←]; M@[B←B,B]; M@[LU←LU,LU];

M@[A←,PF@[ALF@,1]SET[REGIFLAG@,1]];

M@[A←RB,A←];
M@[RB←LU,LR@[1] LU];
M@[RB←A,RB←(LU←A)];
M@[RB←B,RB←(LU←B)];
M@[RB←T,RB←(LU←T)];
M@[RB←RB,RB←(LU←RB)];

M@[B←T,B];
M@[T←LU,LT@[1] LU];
M@[T←A,T←(LU←A)];
M@[T←B,T←(LU←B)];
M@[T←T,T←(LU←T)];
M@[T←RB,T←(LU←RB)];
```

```
%Cycler-masker stuff:

The arguments to the LDF and DISPATCH macros are POS and SIZE, where
POS is the left bit of the field and SIZE the number of bits in the
field.  This is identical to Mesa read-field and write-field
descriptors.


LDF[RBsource,POS,SIZE] is used to right-justify any field.

   0    - 17    20  1-bit fields starting at bit 0, 1, ..., 17
  20    - 36    17  2-bit fields starting at bit 0, 1, ..., 16
  37    - 54    16  3-bit fields starting at bit 0, 1, ..., 15
  55    - 71    15  4-bit fields starting at bit 0, 1, ..., 14
  72    -105    14  5-bit fields starting at bit 0, 1, ..., 13
 106    -120    13  6-bit fields starting at bit 0, 1, ..., 12
 121    -132    12  7-bit fields starting at bit 0, 1, ..., 11
 133    -143    11 10-bit fields starting at bit 0, 1, ..., 10
 144    -153    10 11-bit fields starting at bit 0, 1, ...,  7
 154    -162     7 12-bit fields starting at bit 0, 1, ...,  6
 163    -170     6 13-bit fields starting at bit 0, 1, ...,  5
 171    -175     5 14-bit fields starting at bit 0, 1, ...,  4
 176    -201     4 15-bit fields starting at bit 0, 1, 2, 3
 202    -204     3 16-bit fields starting at bit 0, 1, 2
 205    -206     2 17-bit fields starting at bit 0, 1

DISPATCH[RBsource,POS,SIZE] is used to load APC with the selected
field with SIZE =< 4 bits.

 207    -226    20  1-bit fields starting at bit 0, 1, ..., 17
 227    -245    17  2-bit fields starting at bit 0, 1, ..., 16
 246    -263    16  3-bit fields starting at bit 0, 1, ..., 15
 264    -300    15  4-bit fields starting at bit 0, 1, ..., 14

RSH[RBsource,shiftcount] right-shifts RBsource by shiftcount 1 to 17.

 uses LDF[RBsource,0,(20 - shiftcount)]  codes

LSH[RBsource,shiftcount] left-shifts RBsource by shiftcount 1 to 17.

 301    -317    17 left shifts of 1, ..., 17 bits

LCY[RBsource,shiftcount] left-cycles RBsource by shiftcount 1 to 17.

 320    -336    17 left cycles of 1, ..., 17 bits

RCY[RBsource,shiftcount] right-cycles RBsource by shiftcount 1 to 17.

 uses LCY[RBsource,(20 - shiftcount)]  codes

RHMASK[RBsource] is RBsource & 377

 uses LDF[RBsource,10,10]  code

LHMASK[RBsource] is RBsource & 177400

 337    -337    RBsource & 177400

ZERO is zero

 340    -340   zero
%
```

```
M@[LDF,BS@[3]
IFG@[ADD[#2,#3],20,ER@[Illegal.POS+SIZE],IFG@[1,#3,ER@[Illegal.SIZE],
  FZ@[ADD[#2,IFG@[#3,7,
   SELECT@[SUB[#3,10],133,144,154,163,171,176,202,205],
   SELECT@[#3,100000,0,20,37,55,72,106,121]]]]
  #1]]];

M@[DISPATCH,BS@[3]ALF@[1]SET[RETNOGOOD@,1]
IFG@[#3,4,ER@[Illegal.SIZE],IFG@[ADD[#2,#3],20,ER@[Illegal.POS+SIZE],
  FZ@[ADD[#2,SELECT@[#3,100000,207,227,246,264]]]
  #1]]];

M@[RSH,BS@[3]
IFG@[#2,17,ER@[RSH.count.too.big],IFG@[1,#2,ER@[RSH.count.too.small],
  FZ@[IFG@[#2,7,SELECT@[SUB[#2,10],133,121,106,72,55,37,20,0],
               SELECT@[#2,100000,205,202,176,171,163,154,144]]]
  #1]]];

M@[LSH,BS@[3]
IFG@[#2,17,ER@[LSH.count.too.big],IFG@[1,#2,ER@[LSH.count.too.small],
  FZ@[ADD[#2,300]]
  #1]]];

M@[RCY,BS@[3]
IFG@[#2,17,ER@[RCY.count.too.big],IFG@[1,#2,ER@[RCY.count.too.small],
  FZ@[SUB[337,#2]]
  #1]]];

M@[LCY,BS@[3]
IFG@[#2,17,ER@[LCY.count.too.big],IFG@[1,#2,ER@[LCY.count.too.small],
  FZ@[ADD[#2,317]]
  #1]]];

M@[RHMASK,BS@[3] FZ@[143] #1 ];

M@[LHMASK,BS@[3] FZ@[337] #1 ];

M@[ZERO,BS@[3] FZ@[340] RB];
```

```
%Functions:

Functions are divided into the following classes:

1.  Group A and Group B--only in regular instructions, use F1 and F2.
2.  F1 only--only in regular instructions.
3.  F2 only--either memory reference or regular instructions.
%

*Group A functions are currently unused

*Group B functions
M@[SPAREFUNCTION,FZ@[160] ?];
M@[RESETERRORS,FZ@[161] ?];
M@[INCMPANEL,FZ@[162] ?];
M@[CLEARMPANEL,FZ@[163] ?];
M@[GENSRCLOCK,FZ@[164] ?];
M@[RESETWDT,FZ@[165] ?];
M@[BOOT,FZ@[166] ?];
M@[SETFAULT,FZ@[167] ?];
M@[APCTASK&APC←,FZ@[170]SET[RETNOGOOD@,1] A←];
M@[APC&APCTASK←,FZ@[170]SET[RETNOGOOD@,1] A←];
M@[RESTORE,FZ@[171] A←];
M@[RESETFAULT,FZ@[172] ?];
M@[USECTASK,FZ@[173] ];
M@[WRITECS0&2,FZ@[174] CSX@];
M@[WRITECS1,FZ@[175] CSX@];
M@[READCS,FZ@[176] CSX@];
  M@[CSX@,JC@[6] SET[CSFLG@,1] ER@[CSOp...]];
M@[DOOFF,FZ@[177] ?];

*F1 only
*00 take an RM address as argument
M@[BBFA,REGSHIFT[]PF@[ALF@,3]FF1@[00] #1];        *This is a dispatch
*F1 = 5  is load page
*F1 = 6  is Group A is unused
*F1 = 7  is Group B
M@[RS232←,FF1@[1] B← ];
*02-03 take an RM address as argument
M@[LOADTIMER,FF1@[2] A←#1 ];
M@[ADDTOTIMER,FF1@[3] A←#1 ];
*4 is unused
M@[LOADPAGE,FF1@[5] FF2@[#1] SWPAGE@[1]];
*F1 = 6  is Group A
*F1 = 7  is Group B
*11-14 take an RM address as argument
*F1 = 10  is no-op
M@[WFA,FF1@[11] #1];
M@[BBFB,FF1@[12] #1];
M@[WFB,FF1@[13] #1];
M@[RF,FF1@[14] #1];
M@[BBFBX,FF1@[15] #1];
M@[NEXTINST,FF1@[16] RETCL@[1] ODDCALL@[1] JC@[5] PCF[#1]];      *This is a dispatch
M@[NEXTDATA,FF1@[17] RETCL@[1] ODDCALL@[1] JC@[5] PCF[#1]];
M@[CNEXTDATA,FF1@[17] PCF[#1]]; *Like NextData, but known to be a call, and is to be placed as a call by MicroD

*F2 only
M@[REGSHIFT,FF2@[0] ];
M@[STKP←,FF2@[1] A← ];
M@[FREEZERESULT,FF2@[2]?];
*STACKSHIFT is invisible (used only by STACK, STACK&+1, STACK←, etc.)
M@[STACKSHIFT,FF2@[3]];
M@[IOSTROBE,FF2@[3]]; *same as STACKSHIFT
M@[CYCLECONTROL←,FF2@[4] A← ];
M@[SB←,FF2@[5] A← ];
M@[DB←,FF2@[6] A← ];
M@[NEWINST,ER@[NewInstIsNowByLocation]];
M@[BRANCHSHIFT,FF2@[10]SET[BRSHFLG@,1]];
M@[SALUF←,FF2@[11] B← ];
*12 is a no-op
M@[MNBR←,FF2@[13] A← ];
M@[PCF←,FF2@[14] A← ];
M@[RESETMEMERRS,FF2@[15]];
M@[USECOUTASCIN,FF2@[16]];
M@[PRINTER←,FF2@[17] A← ];
```

```
%Memory reference instructions:

Memory reference clauses are encoded in one of the following forms:
PFETCHn[basereg,raddr,F2];       *n = 1, 2, 4
PSTOREn[basereg,raddr,F2];       *n = 1, 2, 4
IOFETCHn[basereg,device,F2];     *n = 4, 20
IOSTOREn[basereg,device,F2];     *n = 4, 20
XMAP[basereg,raddr,F2];
INPUT[raddr,F2];         *deviceaddr is H2[10,13]orCTASK[0,3],,H2[14,17]
OUTPUT[raddr,F2];        *deviceaddr is H2[10,13]orCTASK[0,3],,H2[14,17]
READPIPE[raddr];
REFRESH[raddr];
The F2 argument is optional.  If given, it causes the displacement to
come from F2 rather than T.

The base register RM address is legal if:
        RAddr is even
        RAddr eq OR@[RAddr,BRBASE]
%
M@[MEM@,SET[ZOT@,IP[#1]]
 IFG@[20,ZOT@,SET[ZOZ@,OR@[ZOT@,BRBASE@]],SET[ZOZ@,ZOT@]]
  IFE@[OR@[ZOZ@,BRBASE@],ZOZ@,
        IFE@[AND@[ZOT@,1],1,ER@[#1.is.an.odd.base.register],]
    IFG@[#2,2,F2@[#3] MRS@[ADD[300,AND@[ZOT@,77]]],
      MRS@[ADD[200,AND@[ZOT@,77]]]],
    ER@[#1.not.addressable.by.task]]];

M@[MEMR@,IFSE@[#1,STACK,SRCDES@[0],SET[ZOT@,IP[#1]]
 IFG@[20,ZOT@,SET[ZOZ@,OR@[ZOT@,BRBASE@]],SET[ZOZ@,ZOT@]]
IFE@[OR@[ZOZ@,BRBASE@],ZOZ@,
IFE@[AND@[ZOT@,#2],0,IFE@[ZOT@,0,ER@[#1.equal.zero..stack.will.be.used.,]]
SRCDES@[ZOT@],ER@[#1.not.even.or.not.quadaligned]],
  ER@[#1.not.addressable.by.task]]]];

M@[IODV@,IFE@[OR@[#1,BRBASE@],#1,SRCDES@[#1],ER@[#1.is.unaddressable.device]]];

*type 0 is unused
M@[IOFETCH4,MEM@[#1,#0,#3] TYPE@[1] IODV@[#2] ?];
M@[READPIPE,MRS@[200] TYPE@[2] MEMR@[#1,0] ?];
M@[REFRESH,TYPE@[3] MRS@[ADD[300,AND@[77,IP[#1]]]]F2@[0] ?];
M@[PFETCH1,MEM@[#1,#0,#3] TYPE@[4] MEMR@[#2,0] ?];
M@[PFETCH2,MEM@[#1,#0,#3] TYPE@[5] MEMR@[#2,1] ?];
M@[PFETCH4,MEM@[#1,#0,#3] TYPE@[6] MEMR@[#2,3] ?];
M@[INPUT,IFG@[#0,1,MRS@[300] F2@[#2],MRS@[200]]
 TYPE@[7] MEMR@[#1] ?];
M@[PSTORE1,MEM@[#1,#0,#3] TYPE@[10] MEMR@[#2,0] ?];
M@[PSTORE2,MEM@[#1,#0,#3] TYPE@[11] MEMR@[#2,1] ?];
M@[PSTORE4,MEM@[#1,#0,#3] TYPE@[12] MEMR@[#2,3] ?];
M@[OUTPUT,IFG@[#0,1,MRS@[300] F2@[#2],MRS@[200]]
 TYPE@[13] MEMR@[#1] ?];
M@[IOFETCH16,MEM@[#1,#0,#3] TYPE@[14] IODV@[#2] ?];
M@[IOFETCH20,MEM@[#1,#0,#3] TYPE@[14] IODV@[#2] ?];
M@[IOSTORE4,MEM@[#1,#0,#3] TYPE@[15] IODV@[#2] ?];
M@[XMAP,MEM@[#1,#0,#3] TYPE@[16] MEMR@[#2,0] ?];
M@[IOSTORE16,MEM@[#1,#0,#3] TYPE@[17] IODV@[#2] ?];
M@[IOSTORE20,MEM@[#1,#0,#3] TYPE@[17] IODV@[#2] ?];
```

```
%Control stuff:
%
*Force absolute location and change default page
M@[AT,@WO@[1] WO@[ADD[#1,#2]] ONPAGE[RSHIFT[ADD[#1,#2],10]]];

*The default micro-instruction

DEFAULT@[IM,(BS@[2] F1@[10] F2@[12] JC@[4] W1@[7777] W2@[7777] PWO@[1])];

*Micro-instructions which are no-ops use the following macro:
M@[NOP,PF@[BS@,2]];

*The ONPAGE macro changes the default page number which is used
*for address assignment.
M@[ONPAGE,DEFAULT@[IM,PGE@[#1]]];

%Branch macros
"~@" and "~" in front of branch condition names are for type checks.
Macros insert these noise characters in front of names supplied by the
program.
%
M@[.-3,ADD[IP[ILC@],-3]];        M@[.+3,ADD[IP[ILC@],3]];
M@[.-2,ADD[IP[ILC@],-2]];        M@[.+2,ADD[IP[ILC@],2]];
M@[.-1,ADD[IP[ILC@],-1]];        M@[.+1,ADD[IP[ILC@],1]];
M@[.,ILC@];

*Regular BC'S:  These conditions result in odd addresses
M@[~ALU#0,CND@[1] JC@[0] JA7@[0] SET[ALUTESTFLG@,1]];
M@[~CARRY,CND@[1] JC@[0] JA7@[1] SET[ALUTESTFLG@,1]];
M@[~ALU<0,CND@[1] JC@[1] JA7@[0] SET[ALUTESTFLG@,1]];
M@[~NOH2BIT8,CND@[1] JC@[1] JA7@[1]];
M@[~R<0,CND@[1] JC@[2] JA7@[0]];
M@[~R ODD,CND@[1] JC@[2] JA7@[1]];
M@[~NOATTEN,CND@[1] JC@[3] JA7@[0]];
M@[~MB,CND@[1] JC@[3] JA7@[1]];

M@[~INTPENDING,CND@[1] JC@[0] JA7@[0] BRANCHSHIFT];
M@[~NOOVF,CND@[1] JC@[0] JA7@[1] SET[ALUTESTFLG@,1] BRANCHSHIFT];
M@[~BPCCHK,CND@[1] JC@[1] JA7@[0] BRANCHSHIFT];
M@[~SPAREBRANCH,CND@[1] JC@[1] JA7@[1] BRANCHSHIFT];
M@[~QUADOVF,CND@[1] JC@[2] JA7@[0] BRANCHSHIFT];
M@[~TIMEOUT,CND@[1] JC@[2] JA7@[1] BRANCHSHIFT];
M@[~,];


*Complementary BC's:  These conditions result in even addressed
M@[~@ALU=0,CND@[1] JC@[0] JA7@[0] SET[ALUTESTFLG@,1]];
M@[~@NOCARRY,CND@[1] JC@[0] JA7@[1] SET[ALUTESTFLG@,1]];
M@[~@ALU>=0,CND@[1] JC@[1] JA7@[0] SET[ALUTESTFLG@,1]];
M@[~@H2BIT8,CND@[1] JC@[1] JA7@[1]];
M@[~@R>=0,CND@[1] JC@[2] JA7@[0]];
M@[~@R EVEN,CND@[1] JC@[2] JA7@[1]];
M@[~@IOATTEN,CND@[1] JC@[3] JA7@[0]];
M@[~@NOMB,CND@[1] JC@[3] JA7@[1]];

M@[~@NOINTPENDING,CND@[1] JC@[0] JA7@[0] BRANCHSHIFT];
M@[~@OVF,CND@[1] JC@[0] JA7@[1] SET[ALUTESTFLG@,1]BRANCHSHIFT];
M@[~@BPCNOCHK,CND@[1] JC@[1] JA7@[0] BRANCHSHIFT];
M@[~@SPARENOBRANCH,CND@[1] JC@[1] JA7@[1] BRANCHSHIFT];
M@[~@INQUAD,CND@[1] JC@[2] JA7@[0] BRANCHSHIFT];
M@[~@NOTIMEOUT,CND@[1] JC@[2] JA7@[1] BRANCHSHIFT];
M@[~@,];

M@[DBAT@,IDF@[~@#3,W1@[#1] W2@[#2](~@#3,~@#4),W2@[#1] W1@[#2](~#3,~#4)]];
M@[BAT@,IDF@[~@#2,W1@[#1] (~@#2,~@#3),W2@[#1] (~#2,~#3)]];
M@[GOTOX@,W1@[#1] IFE@[CSFLG@,1,JC@[6],JC@[4]]];

%Branch and Goto
The branch and goto macros are now identical and are
interchangeable.  If the next micro-instruction to be executed
is in a different page, the macro must have a P following it.
Thus GOTOP[xyz] is used when xyz is in a different page from
the current micro-instruction.  This occurs only when the m-i
preceding the current one does a LOADPAGE.
%

M@[DBLGOTO,DBAT@[#1,#2,#3,#4]];
M@[DBLGOTOP,CHPAGE@[1] DBAT@[#1,#2,#3,#4]];
M@[DBLBRANCH,DBAT@[#1,#2,#3,#4]];
M@[DBLBRANCHP,CHPAGE@[1] DBAT@[#1,#2,#3,#4]];

M@[GOTO,IFSE@[#2#3,,GOTOX@[#1],
 BAT@[#1,#2,#3]]?];
M@[GOTOP,IFSE@[#2#3,,GOTOX@[#1] CHPAGE@[1],
 BAT@[#1,#2,#3]]?];
M@[BRANCH,IFSE@[#2#3,,GOTOX@[#1],
 BAT@[#1,#2,#3]]?];
M@[BRANCHP,IFSE@[#2#3,,GOTOX@[#1] CHPAGE@[1],
 BAT@[#1,#2,#3]]?];
M@[SKIP,GOTO[.+2,#1]];

*********** External References
M@[GOTOEXTERNAL,IFSE@[#2#3,,
  RETCL@[2] JC@[4] JA@[AND@[#1,377]],
  ER@[No.conditional.external.goto]]];
M@[CALLEXTERNAL,IFSE@[#2#3,,
  W2@[.+1] RETCL@[3] JC@[5] JA@[AND@[#1,377]],
  ER@[no.args.allowed.in.external.call's]]];
M@[LOADPAGEEXTERNAL,FF1@[5] FF2@[#1]];
```

```
%Calls must normally be executed from even locations, because
the return is to the caller's address or 1.  MicroD will only place calls at even word locations.  The NEXTINST, NEXTop, and NEXTDATA
macros are required to be calls from odd locations to aid the
instruction buffer refill micro-code in re-execution following
loading of the buffer.  The macro RCALL[label] or RCALL will
cause a call to be assigned to an odd location.
%

M@[CALL,IFSE@[#2#3,,RETCL@[1] JC@[5] W1@[#1] W2@[.+1],
ER@[no.args.allowed.in.call's]]?];
M@[CALLP,IFSE@[#2#3,,RETCL@[1] CHPAGE@[1] JC@[5] W1@[#1] W2@[.+1],
ER@[no.args.allowed.in.callp's]]?];
M@[RCALL,IFSE@[#2#3,,RETCL@[1] ODDCALL@[1] JC@[5] IFSE@[#1,,W1@[.+1],W1@[#1]],
ER@[no.args.allowed.in.rcall's]]?];
M@[RCALLP,IFSE@[#2#3,,RETCL@[1] ODDCALL@[1] CHPAGE@[1] JC@[5] IFSE@[#1,,W1@[.+1],W1@[#1]],
ER@[no.args.allowed.in.rcallp's]]?];

M@[TASK,W1@[.+1] W2@[.+2] SET[TSKFLG@,1] JC@[5] RETCL@[1]];

M@[RETURN,IFSE@[#1#2,,RETCL@[2] JC@[6] JA7@[0],
ER@[no.args.allowed.in.return's]]?];

M@[NOTASKRTN,IFSE@[#1#2,,RETCL@[2] JC@[6] JA7@[0],
ER@[no.args.allowed.in.notaskrtn's]]?];

M@[NIRET,IFSE@[#1#2,,RETCL@[2] JC@[6] JA7@[1],
ER@[no.args.allowed.in.return's]]?];

M@[DISP,IFSE@[#2#3,,JC@[7] W1@[#1],
ER@[no.args.allowed.in.disp's]]?];

M@[DISPP,IFSE@[#2#3,,JC@[7] CHPAGE@[1] W1@[#1],
ER@[no.args.allowed.in.dispp's]]?];
```

```
%ALU stuff:

Alu operations must be defined for (A, RB) op (B,T)
%

M@[LU←B,ALF@[0]LU];
M@[LU←T,ALF@[0]LU];
M@[LU←RB,ALF@[1]LU];
M@[LU←A,ALF@[1]LU];
M@[RBANDB,ALF@[2]LU];
M@[AANDB,ALF@[2]LU];
M@[RBANDT,ALF@[2]LU];
M@[AANDT,ALF@[2]LU];
M@[RBORB,ALF@[3]LU];
M@[AORB,ALF@[3]LU];
M@[RBORT,ALF@[3]LU];
M@[AORT,ALF@[3]LU];
M@[RBXORB,ALF@[4]LU];
M@[AXORB,ALF@[4]LU];
M@[RBXORT,ALF@[4]LU];
M@[AXORT,ALF@[4]LU];
M@[RB#B,ALF@[4]LU];
M@[A#B,ALF@[4]LU];
M@[RB#T,ALF@[4]LU];
M@[A#T,ALF@[4]LU];
M@[RBANDNOTB,ALF@[5]LU];
M@[AANDNOTB,ALF@[5]LU];
M@[RBANDNOTT,ALF@[5]LU];
M@[AANDNOTT,ALF@[5]LU];
M@[RBORNOTB,ALF@[6]LU];
M@[AORNOTB,ALF@[6]LU];
M@[RBORNOTT,ALF@[6]LU];
M@[AORNOTT,ALF@[6]LU];
M@[RBXNORB,ALF@[7]LU];
M@[AXNORB,ALF@[7]LU];
M@[RBXNORT,ALF@[7]LU];
M@[AXNORT,ALF@[7]LU];
M@[RB=B,ALF@[7]LU];
M@[A=B,ALF@[7]LU];
M@[RB=T,ALF@[7]LU];
M@[A=T,ALF@[7]LU];
M@[RB+1,ALF@[10]LU];
M@[A+1,ALF@[10]LU];
M@[RB+B,ALF@[11]LU];
M@[A+B,ALF@[11]LU];
M@[RB+T,ALF@[11]LU];
M@[A+T,ALF@[11]LU];
M@[RB+B+1,ALF@[12]LU];
M@[A+B+1,ALF@[12]LU];
M@[RB+T+1,ALF@[12]LU];
M@[A+T+1,ALF@[12]LU];
M@[RB-1,ALF@[13]LU];
M@[A-1,ALF@[13]LU];
M@[RB-B,ALF@[14]LU];
M@[A-B,ALF@[14]LU];
M@[RB-T,ALF@[14]LU];
M@[A-T,ALF@[14]LU];
M@[RB-B-1,ALF@[15]LU];
M@[A-B-1,ALF@[15]LU];
M@[RB-T-1,ALF@[15]LU];
M@[A-T-1,ALF@[15]LU];

*ALUF[16] UNASSIGNED

M@[RBSALUFOPB,ALF@[17]LU];
M@[ASALUFOPB,ALF@[17]LU];
M@[RBSALUFOPT,ALF@[17]LU];
M@[ASALUFOPT,ALF@[17]LU];
```

```
*Macro executed after assembling instruction to default W1
SETPOST@[IM,IMX@];
M@[IMX@,SET[CSFLG@,0]
  IFE@[CHPGFLG@,1,CHPAGE@[1] SET[CHPGFLG@,0]]
  IFE@[RTNFLG@,1,XX1@[]]
  IFE@[TSKFLG@,1,XX2@[]]
  IFA@[SWPAGE@,IFA@[W1@,,SET[CHPGFLG@,1]]]
  IFE@[REGIFLAG@,1,XX3@[]]
  IFE@[RETNOGOOD@,1,XX4@[]]
  IFE@[RTNTOFLG@,1,XX5@[]]
  SET[BRSHFLG@,0]SET[ALUTESTFLG@,0] ];
M@[XX1@,JC@[6] JA7@[0] RETCL@[2] SET[RTNFLG@,0]
   SET[RTNTOFLG@,1]];
M@[XX2@,SET[RTNFLG@,1] SET[TSKFLG@,0]];
M@[XX3@,IFE@[NOILKOKFLG@,0,IFE@[FVAL@[ALF@],0,
 ER@[WARNING:..no.register.interlock]]SET[REGIFLAG@,0]SET[NOILKOKFLG@,0]]];
M@[XX4@,IFE@[FVAL@[JC@],6,
   ER@[ERROR:..apc.loaded.during.return]] SET[RETNOGOOD@,0]];
M@[XX5@,IFE@[ALUTESTFLG@,1,
   ER@[ERROR:..alu.results.tested.following.return]] SET[RTNTOFLG@,0]];
M@[NOREGILOCKOK,SET[NOILKOKFLG@,1]];

%"TITLE" outputs the file name and the value of ILC on the .ER file
to help correlate error messages with source statements.  It also resets
various assembly flags to standard states.
%
M@[TITLE,(SETTASK[0] DIB@[] TARGET@[ILC@]
  SET[TSKFLG@,0] SET[RTNFLG@,0] SET[CSFLG@,0] SET[CHPGFLG@,0]
  SET[REGIFLAG@,0] SET[RTNTOFLG@,0] SET[RETNOGOOD@,0]
  SET[NOILKOKFLG@,0] MIDASINIT[])];

M@[MIDASINIT,IFE@[INITFLG@,0,SET[INITFLG@,1]
  IMRESERVE[0,0,2]IMRESERVE[0,100,21]IMRESERVE[17,0,400]
  LANGVERSION[],]];

M@[DIB@,IFE@[MULTDIBFLG@,0,SET[MULTDIBFLG@,1]
  VERSION[V@,0] V@[(VERS@[1])]]];

M@[NOMIDASINIT,SET[INITFLG@,1]];

M@[LANGVERSION,SETTASK[17]RV[D0LANGVERSION,41,3]SETTASK[0]];

M@[MULTDIB,SET[MULTDIBFLG@,1]];

M@[END,ER@[END...ILC=,0,IP[ILC@]]];

SET[INITFLG@,0];

SET[MULTDIBFLG@,0];

*If mode not defined, make it Pilot
IDF@[AltoMode,,SET[AltoMode, 0]];

*This statement defines comments *// is Alto and *= is Pilot
IFE@[AltoMode,0,COMCHAR@[//],COMCHAR@[=]];

*Print Message telling Mode
IFE@[AltoMode,0,ER@[Pilot.3.0.Microcode],ER@[Alto/Mesa.5.0.D0.Microcode]];

*Macros
M@[OPCODE, AT[2001,LSHIFT[#1,2]]];
*Cycler-masker functions
M@[FIXVA, BS@[3] FZ@[341] #1];
M@[Form1,LDF[#1,17,1]];
M@[Form2, BS@[3] FZ@[342] #1];
M@[Form3,LDF[#1,16,2]];
M@[Form4, BS@[3] FZ@[343] #1];
M@[FormMinus4, BS@[3] FZ@[351] #1];


M@[$RSetDispLo,  AND@[ADD[LSHIFT[#1,14],#2],377]C];
M@[$RSetDispHi,  AND@[ADD[LSHIFT[#1,14],#2],177400]C];

ER@[6/12/79--D0Lang.version.4]; *Print release date on .ER file
```

```
	title[dmdefs];
*	last modified by Jarvis March 15, 1979  4:10 PM

*DEFS FOR DISK MOD 31

SET TASK[DTASK];

* I/O register assignments
SET[DRUNO, LSHIFT[DTASK, 4]];
SET[OPUT, ADD[DRUNO, 5]];
SET[IPUT, ADD[DRUNO, 3]];
SET[DSKADD, 6];			* DISKADD REG ← ODATA
SET[DBUFFER, 5];		* KOB ← ODATA
SET[DPRP, 4];			* PRP ← ODATA
SET[DPWP, 3];			* PWP ← ODATA
SET[DSKCMMD, 2];		* CCSR ← ODATA
SET[DSKCTRLB, 1];		* DISK CTRLB ← ODATA
SET[DSKCTRLA, 0];		* DISK CTRLA ← ODATA
SET[DID, 0];			* IDATA ← DISK ID
SET[DRPWP, 1];			* IDATA ← PWP/PRP/KOBST/KIBST
SET[DSTATUS, 2];		* IDATA ← DISK STATUS
SET[DKIB, 3];			* IDATA ← KIB (DISK READ BUFFER)

* R store assignments
SET[DRBASE, AND@[60, DRUNO]];			* this tasks registors start here
RV[KData, ADD[DRBASE, 2]];		* WRITE MEMORY ADRESS
RV[KData1, ADD[DRBASE, 3]];
RV[RP, ADD[DRBASE, 4]];				* READ POINTER FOR DISK
RV[WP, ADD[DRBASE, 5]];				* WRITE POINTER FOR DISK
RV[READMEMCOUNT, ADD[DRBASE, 6]];		* READ MEMORY COUNT OF WORDS
RV[WRITEMEMCOUNT, ADD[DRBASE, 7]];		* WRITE MEMORY COUNT OF WORDS
RV[CP, ADD[DRBASE, 10]];			* CHECK OPERATION PING PONG POINTER
RV[RPP, ADD[DRBASE, 11]];			* Flags see below
RV[DCBADDRESS, ADD[DRBASE, 12]];		* DCB ADDRESS
RV[DCBADDRESS1, ADD[DRBASE, 13]];
RV[KBADDRESS, ADD[DRBASE, 14]];			* DCBADDRESSLOCK ADDRESS
RV[KBADDRESS1, ADD[DRBASE, 15]];
RV[COMMANDWORD, ADD[DRBASE, 16]];		* DCB COMMAND WORD
RV[WR, ADD[DRBASE, 17]];			* WORKING REGESTER

* beware these registers are recycled
RV[DISKSTATUS, ADD[DRBASE, 2]];			* STATUS WORD
RV[OLDDISKADD, ADD[DRBASE, 4]];			* LAST SEEK POSITION
RV[SECTORWAKEUP, ADD[DRBASE, 5]];		* SECTOR MASK
RV[PROCESSWAKEUPREG1, ADD[DRBASE, 6]];
RV[DCB, ADD[DRBASE, 7]];			* POINTER TO NEXT DCB
RV[NEWDISKADD, ADD[DRBASE, 10]];		* NEW SEEK POSITION

SET[DRCBASE,  LSHIFT[DRPAGE,10]];  * control store base address of DRPAGE
SET[DCommandBase, ADD[DRCBASE, 100]];	* read, write, check field dispatch
SET[DPingBase, ADD[DRCBASE, 120]];	* ping pong dispatch
MC[DTASK.1, LSHIFT[DTASK, 14]];
MC[drpage.2, DRCBASE];

* RPP flags, context sensitive semantics for finding and transferring sector
MC[Seeking, 1];	* seek in progress (note that same bit used for label flag)
MC[LabelFlag, 1]; * processing label during sector transfer
MC[PostIOCB, 2];  * write status into IOCB/processing data field in transfer
		  * RPP < 0 implies disk error
		  * RPP = 0 implies processing header during sector transfer

* CP states
MC[ReadPing, 0];
MC[WritePing, 1];
MC[CPRead, 2];
MC[CPCheck, 3];
MC[CPWrite, 4];

* CTRLA functions
MC[SelectiveReset, 1];
MC[ResetWakeup, 2];
MC[SetStrobe, 4];
MC[ClearStatus, 10];
MC[ResetEverything, 13];

* CTRLB function - assumed that WakeupAllow is always on
MC[WakeupAllow, 1];
MC[DataTaskAllow, 3];
MC[SendDiskAddress, 11];
MC[WakeMeEveryWord, 23];
MC[WakeMeForOutput, 43];

* Status register
MC[SeekInProgress, 100];
MC[SeekFailed, 200];
MC[DataWake, 400];
MC[SectorWake, 1000];
MC[SectorMask, 170000];  * same for IOCB

* CMMD
MC[SetSeekIdle, 1];
MC[ReadHLD, 250];
MC[HeaderLabelDataMask, 374];	* same for IOCBCommand

* CSB offsets
SET[CSBNext, 0];
SET[CSBStatus, 1];
SET[CSBDiskAdr, 2];
SET[CSBSectorMask, 3];

* IOCB word offsets
```

```
SET[IOCBNext, 0];
SET[IOCBStatus, 1];
SET[IOCBCommand, 2];
SET[IOCBHeaderPointer, 3];
SET[IOCBLabelPointer, 4];
SET[IOCBDataPointer, 5];
SET[IOCBTransferMask, 6];
SET[IOCBErrorMask, 7];
SET[IOCBUnused, 10];   * 11 for extended address format.
MC[IOCBDiskAdr, 11];   * 12 for extended address format
SET[IOCBAddressExtend, 10];   * high order bits of label and data address
    M@[$LabelExtend, LDF[#1, 0, 10]];
    M@[$DataExtend, LDF[#1, 10, 10]];


* IOCB status
MC[IOCBStatusValid, 7400];
MC[IOCBTransferOk, 0];
MC[IOCBHardwareError, 1];
MC[IOCBCheckError, 2];
MC[IOCBIllegalSector, 3];

* IOCB disk address
MC[IOCBDrive, 2];

* IOCB command
MC[SeekOnly, 2];
MC[DriveModifier, 1];
MC[IOCBRead, 0];  * 2 bit command field for header, label, and data
MC[IOCBCheck, 1];
MC[IOCBWrite, 2];
MC[IOCBWrite1, 3];
M@[$Seal, LDF[COMMANDWORD, 0, 10]];  * field extractor for seal
M@[$ExtendBit, LDF[COMMANDWORD, 7, 1]]; * field extractor for extend bit
MC[ValidSeal, 110];  * Seal=110 alto style dcb, Seal=111 extended address IOCB
MC[ExtendHint, 1];

* physical characteristics of the disk format
MC[SectorsPerTrack, 14];
MC[HeaderSize, 2];
MC[LabelSize, 10];
MC[DataSize, 400];

* disk address field extractors
M@[$Sector, LDF[#1, 0, 4]];
M@[$Cylinder, LDF[#1, 4, 11]];
M@[$Track, LDF[#1, 4, 13]];  * includes cylinder, head, and disk
M@[$Disk, LDF[#1, 16, 1]];
```

```
insert[d0lang];
NOM[DASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[DMDefs];
        TITLE[DMInit];
*last modified by Johnsson on April 7, 1979  12:30 PM
*last modified by CPT on March 15, 1979  12:21 PM

        ON PAGE[DiskInitPage];
        settask[DTASK];

DiskInit: WR←13C, at[DiskInitLoc];
        OUTPUT[WR, DSKCTRLA];
        KData← 0C;
        KData1← 0C;
        RP← 0C;
        WP← 0C;
        ReadMemCount← 0C;
        WriteMemCount← 0C;
        CP← 0C;
        RPP← 0C;
        DCBADDRESS← 0C;
        DCBADDRESS1← 0C;
        CommandWord← 0C;
        WR← 1C;
        OUTPUT[WR,DSKCTRLB]; *ALLOW WAKE UPS
        KBADDRESS← 400C; *KBaddress ← 521b
        KBADDRESS← (KBADDRESS)+(121C);
        KBADDRESS1← 0C;  *build pointer to kblock
        loadpage[drpage] ;
        GOTOp[DSWTASK];*will set tpc and return


        end[dminit];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[DMDefs];
        TITLE[extended-address-DMTask];
*       last edit by Johnsson on April 7, 1979  12:31 PM
*       second page elim. by Johnsson on March 15, 1979  5:40 PM
*       last modified by Jarvis on March 15, 1979  12:07 PM
*       DoInt added by Johnsson on February 16, 1979  12:53 AM

ON PAGE[DRPAGE];
settask[DTASK];
M@[ONES,(ZERO)-1];

* sector wake up point if not transfering data
DSWTASK:
        call[cReturn]; * allow task switch before subsequent storage reference
        PFETCH1[KBADDRESS, DCBADDRESS, CSBNext];  * get the link to the dcb
DSWA1:  PROCESSWAKEUPREG1←0C;  * clear my interrupt regester
        T ← IOCBDiskAdr, TASK;  * set up for fetch of IOCBDiskAdr
        INPUT[DISKSTATUS, DSTATUS];  * get disk status, need sector information
        DISKSTATUS←(DISKSTATUS) XOR (SectorMask);  * sector complemented
        DISKSTATUS←(DISKSTATUS) OR (IOCBStatusValid);  * put in the 17
        PFETCH1[DCBADDRESS, COMMANDWORD, IOCBCommand];  * get the disk command
DSWA2:  LU←(RPP) XOR (Seeking);
        GOTO[SeekService, ALU=0], LU←DCBADDRESS;  * jump for seek in progress
        GOTO[SectorAll, ALU=0], WR ← ClearStatus;  * jump for NIL IOCB pointer
                                                   * set WR for possible errors
* process IOCB
        T ← ($ExtendBit) + (T); * new format IOCBDiskAdr displaced by one
        PFETCH1[DCBADDRESS, NEWDISKADD];  * get the new seek info
        TASK, PFETCH1[KBADDRESS, OLDDISKADD, CSBDiskAdr];  * get old position
        T←ValidSeal;
        T←($Seal) XOR (T);
        KData1 ← T;  * non-zero implies extended addressing
        GOTO[.+2, ALU#0], LU ← (KData1) XOR (ExtendHint);
          GOTO[.+2], T←SectorsPerTrack;  * Seal=ValidSeal
        GOTO[InvalidSeal, ALU#0], T←SectorsPerTrack;  * jump for invalid seal
        LU←($Sector[NEWDISKADD])-(T);
        GOTO[DSWL, ALU>=0], WR←3C;    * jump for illegal sector, WR ← garbage?
        lu ← commandword, goto[.+2, r even];  * test DriveModifier
         newdiskadd ← (newdiskadd) xor (IOCBDrive);
        nop;  * call cannot be even target of condtional
        TASK, PSTORE1[KBADDRESS, NEWDISKADD, CSBDiskAdr];
        T←$Track[OLDDISKADD];            * Is old track
        LU←($Track[NEWDISKADD]) XOR (T);  * . . . the same as new track?
        GOTO[MoveArm, ALU#0], LU←(COMMANDWORD) AND (SeekOnly);
        GOTO[JustSeek, ALU#0], T←$Sector[DISKSTATUS];  * jump for seek only
        T←($Sector[NEWDISKADD]) XOR (T), GOTO[DSWJ6, IOATTEN];
        DBLGOTO[SectorAll1, AtSector, ALU#0], RPP←0C;   * at the right sector?

* sector wakeup clean up
SectorAll:  NOP;
SectorAll1: WR←SetSeekIdle;
        OUTPUT[WR, DSKCMMD], GOTO[DSWJ7, IOATTEN];  * SET SEEK IDLE COMMAND
        NOP;
        NOP;*two m-i after outputs

* process sector interrupt mask
        PFETCH1[KBADDRESS, SECTORWAKEUP, CSBSectorMask], TASK;
        * allow task switch before preceeding storage reference interlocks
        T←PROCESSWAKEUPREG1;
        T ← SECTORWAKEUP←(SECTORWAKEUP)OR(T);* SECT INTERUPT MASK
** checking IOATTEN causes endless loop **
        PSTORE1[KBADDRESS, DISKSTATUS, CSBStatus];  * set current disk status
** warning no tasking allowed from here to the last pstore before DSWC2 **
        loadpage[0];  * get bits to OR into NWW
        callp[DoInt];  * set NWW and IntPending; uses regs 0,1; no task

* get link to next DCB
        PFETCH1[DCBADDRESS, DCB, IOCBNext], CALL[IOTask];
        LU ← RPP, GOTO[DSWCX, R>=0];  * Check for error
          DCB ← (DCB) and (0C);  * error, don't chain  * insure read comp
          WR←ones;
          PSTORE1[KBADDRESS, WR, CSBDiskAdr], CALL[IOTask];
          RPP ← PostIOCB;
DSWCX:  LU ← (RPP) XOR (PostIOCB);  * check data loop or seek command
        GOTO[DSWC2, ALU#0];
        LU←DCBADDRESS;
        GOTO[DSWC4, ALU=0], LU←DCBADDRESS;

* just completed IOCB, post status and cdr the IOCB chain
        PSTORE1[KBADDRESS, DCB, CSBNext];  * new DCB pointer to kblock
        PSTORE1[DCBADDRESS, DISKSTATUS, IOCBStatus], CALL[IOTask];  * into dcb
        NOP;*why is this nop here???
        T ← DCB;  * Chain to new DCB
        DCBADDRESS ← T;  * proceed if something there
DSWC4:  RPP←0C, GOTO[DSWA1, ALU#0];  * jump to process next IOCB
        NOP;  * end of IOCB chain
DSWC2:  WR←ResetWakeup;
        OUTPUT[WR, DSKCTRLA];  * RESET SECTOR WAKEUP
        WR←WR, GOTO[DSWTASK];  * INTERLOCK!!!!

* seek only  * NO ERROR interrupt
JustSeek:  PFETCH1[DCBADDRESS, PROCESSWAKEUPREG1, IOCBTransferMask];
        RPP←PostIOCB, GOTO[SectorAll1];

* seek in progress
SeekService:  LU ← (DISKSTATUS) AND (SeekFailed);
        GOTO[SeekFailure, ALU#0], LU ← (DISKSTATUS) AND (SeekInProgress);
        GOTO[.+2, ALU#0];
```

```
          RPP←0C, GOTO[DSWA2];  * seek done
        GOTO[SectorAll1];

* set arm in motion
MoveArm:  T←$Disk[OLDDISKADD];  * SEE IF WE ARE CHANGING DISKS
        LU←($Disk[NEWDISKADD]) XOR (T);
        WR←SetSeekIdle, GOTO[SwitchDrive, ALU#0];  * jump to switch drives
        OUTPUT[NEWDISKADD, DSKADD];  * OUTPUT NEW SEEK INFO
        WR←SendDiskAddress, TASK;
        T←$Cylinder[NEWDISKADD];  * CHECK FOR CHANGE IN TRACK ADDRESS
        LU←($Cylinder[OLDDISKADD]) XOR (T);
        GOTO[.+2, ALU#0], RPP←Seeking;
         PSTORE1[KBADDRESS, NEWDISKADD, CSBDiskAdr], GOTO[DSWTASK]; * hd switch
        OUTPUT[WR, DSKCTRLB];       * SET SEND DISK ADDRESS
        WR←SetStrobe;
        OUTPUT[WR, DSKCTRLA]; * set strobe
        GOTO[SectorAll], RPP←Seeking;        * SET SEEK flag

* switch drives
SwitchDrive: OUTPUT[WR, DSKCMMD];  * SET SEEK IDLE COMMAND
        WR←ResetWakeup;
        OUTPUT[WR, DSKCTRLA];  * RESET SECTOR WAKEUP
        OUTPUT[NEWDISKADD, DSKADD];  * DISK CHANGE OVER
        WR←ones;
        T←WR←(WR) XOR (IOCBDrive);  * don't clobber drive
        NEWDISKADD←(NEWDISKADD) OR (T);
        PSTORE1[KBADDRESS, NEWDISKADD, CSBDiskAdr];
        GOTO[DSWTASK], RPP←Seeking;  * WE WILL RE-DO SEEK AFTER DISK CHANGEOVER
```

```
* on right sector, prepare for transfer
AtSector: nop;
        TASK, PFETCH1[DCBADDRESS, KData, IOCBHeaderPointer];

*translate alto style read/check/write into IRDC style
* The commands for each field are as follows
* Alto function  IRDC    Let xa = high order bit of alto function
*   00   read     10     Let ya = low order bit of alto function
*   01   check    11     Let xi = high order bit of IRDC function
*   11   write    01     Let yi = low order bit of IRDC function
*   10   write    01        then xi = ~xa and yi = xa + ya

* The old method uses 24. words of code and executes 6 instruction to do the
* command translation.  Notice with awe that the new method uses but 6 words
* of code and excutes 6 instructions.
         T ← (COMMANDWORD) AND (HeaderLabelDataMask);    * get command field
         WR ← T;
         WR ← (RSH[WR, 1]) OR (T);        * shift high bits over low
         WR ← (WR) AND (124C);            * low order bits
         task,T ← (COMMANDWORD) ornot (250C);
         WR ← (WR) ornot (T);

        RP←0C;
        T←HeaderSize, CALL[SetShortFieldConstants];
        WP←0C, GOTO[DSWJ10, IOATTEN];
        OUTPUT[WR, DSKCMMD];        * output command to disk
ProcessField: CP←ResetWakeup;
        OUTPUT[CP, DSKCTRLA];   * reset sector wakeup
        WR←DataTaskAllow;
        OUTPUT[WR, DSKCTRLB];   * set DATA TASK ALLOW
        CP←CPRead;  * assume read
        DISPATCH[COMMANDWORD, 10, 2];
        DISP[.+1];  * header, label and data all use this dispatch
      READMEMCOUNT←0C, GOTO[FieldAll],        AT[DCommandBase, IOCBRead!];
      WR←WakeMeForOutput, GOTO[CheckField], AT[DCommandBase, IOCBCheck!];
      WR←WakeMeForOutput, GOTO[WriteField], AT[DCommandBase, IOCBWrite!];
      WR←WakeMeForOutput, GOTO[WriteField], AT[DCommandBase, IOCBWrite1!];

* check field on disk drive
CheckField:  OUTPUT[WR, DSKCTRLB];
        CP←CPCheck, GOTO[FieldAll];       * FOR  PRELOAD

* write field on disk
WriteField: OUTPUT[WR, DSKCTRLB];
        CP←CPWrite;
        WRITEMEMCOUNT←0C;
        WR←1C;
****warning no tasking allowed between the two outputs for buffer
        OUTPUT[WP, DPWP];           * OUTPUT WRITE POINTER
        OUTPUT[WR, DBUFFER], GOTO[DSWJL12, IOATTEN];  * OUTPUT SINC WORD
        WP←(WP)+1;

* read, write, and check all flow through here
FieldAll: T← KData, call[cReturn]; * allow task switch
         DISPATCH[CP, 15, 3], GOTO[DSWJL, IOATTEN];
         WR← T, DISP[.+1]; * set WR for storage alignment calculation
           CP←WritePing, GOTO[ReadDisk],   AT[DPingBase, ReadPing!];
           CP←ReadPing, GOTO[WriteDisk],   AT[DPingBase, WritePing!];
           CP←CPRead, GOTO[ReadDisk],      AT[DPingBase, CPRead!];
           T←366C, GOTO[DoCheck],          AT[DPingBase, CPCheck!];
           CP←CPWrite, GOTO[WriteDisk],    AT[DPingBase, CPWrite!];

* Check field of sector
* Microcode uses write logic, but hardware does not actually write on disk.
* Both header and label fit in the 20 word hardware buffer.  The header and
* label code first copies the entire field into the buffer using the main
* write loop.  The hardware then reads the disk and leaves the result of the
* check operation in the buffer.  The code then writes the results back into
* the central store by falling into the main read loop.

* The microcode tries to win on the data field, but the algorithm appears
* buggy.  Note that CP is set to CPRead when control falls out of the main
* output loop.  This prevents control from passing back through DoCheck.
DoCheck:   LU←(READMEMCOUNT)-(T);  * can't check too large a block of data
         LU←(RPP) XOR (PostIOCB), GOTO[WriteDisk, ALU>=0];
         CP←ReadPing, GOTO[.+2, ALU=0];  * bug? eventually resets CP to CPRead
           CP←CPWrite, GOTO[WriteDisk];  * header or label
         WR←DataTaskAllow;  * processing data field
         OUTPUT[WR, DSKCTRLB];
         WR←(WR), GOTO[FieldAll];  * STOP KOB BUFFER

* output loop (read store and send data to the disk kob buffer)
* [if quad align and wc>4 do iofetch4 else do output one]
****** warning no tasking allowed between the output wp and the buffer command
WriteDisk: T ← ReadMemCount;
         WR ← (WR)+T;  * WR has a copy of KData
         LU←(WR) AND (3C);  * CHECK QUAD BOUND
         GOTO[DDWP1, ALU#0], LU←(ReadMemCount)-(4C);
         GOTO[DDWP, ALU<0], T ← (ReadMemCount)-1;

* quad word write to disk works here
         WP←(WP)+(3C);  * next instruction allows write of WP before the output
         ReadMemCount←(ReadMemCount)-(3C), call[cReturn];  * CHECK COUNT >3
         OUTPUT[WP, DPWP];                          * OUTPUT WRITE POINTER
         nop;               * two m-i after output
         T ← (ReadMemCount)-1;
         IOFETCH4[KData, OPUT], GOTO[DDWV];          * OUTPUT 4 WORDS

* write a single word onto disk
DDWP1:  T ← (ReadMemCount)-1;
```

```
DDWP:   PFETCH1[KData, WR];                  * GET ONE FROM MEMORY
        OUTPUT[WP, DPWP];                    * OUTPUT WRITE POINTER
        OUTPUT[WR, DBUFFER];                 * OUTPUT ONE
DDWV:   ReadMemCount←(ReadMemCount)-1;       * DOWN COUNT WORD COUNTER
        GOTO[.+2, ALU=0], WP←(WP)+1;         * jump for end of field
          GOTO[FieldAll];
        CP←CPRead;  * STOP WRITING
        WR←DataTaskAllow;
        OUTPUT[WR, DSKCTRLB];  * STOP KOB BUFFER
        LU←WRITEMEMCOUNT, GOTO[DDWU1];

* input loop (we read kib buffer and send data to the main store)
* if quad align and wc>4 do then iostore4 else do input one
* if wc<4 set force wakeup on one, in the disk  and set cp = to one for
* termination
******warning no tasking allowed between the output rp and the buffer command
ReadDisk: T ← WriteMemCount;
        WR ← (WR)+T;  * WR has a copy of KData
        LU←(WR) AND (3C);  * CHECK QUADBOUND
        GOTO[DDWO, ALU#0], LU←(WriteMemCount)-(4C);
        GOTO[DDWO1, ALU<0], T ← (WriteMemCount)-1;    * CHECK COUNT >3

* quad word read from disk works here
        WriteMemCount←(WriteMemCount)-(3C);
        RP←(RP)+(3C), call[cRETURN];
        OUTPUT[RP, DPRP];                               * OUTPUT READ POINTER
        T ← (WriteMemCount)-1;
        RP←(RP)+1;  * invoke interlock so that pointer ok before data arrives
        IOSTORE4[KData, IPUT], GOTO[DDWU];              * INPUT 4 WORDS

* read a single word from disk
DDWO:   T ← (WriteMemCount)-1;
DDWO1:  OUTPUT[RP, DPRP];                               * OUTPUT READ POINTER
        RP←(RP)+1;  * invoke interlock so that pointer ok before data arrives
        INPUT[WR, DKIB];                                * INPUT ONE TO WR
        PSTORE1[KData, WR];                             * PUT IT IN MEMORY
DDWU:   WriteMemCount←(WriteMemCount)-1;                * DOWN COUNT WORD COUNTER
DDWU1:  GOTO[EndField, ALU=0], LU←RPP;
        LU←RSH[WriteMemCount, 2];
        GOTO[.+2, ALU#0], WR←WakeMeEveryWord;
          OUTPUT[WR, DSKCTRLB], GOTO[FieldAll];
        GOTO[FieldAll];

* dispatch for next field
EndField: GOTO[StartLabel, ALU=0], LU←(RPP) XOR (LabelFlag);
        GOTO[StartData, ALU=0], RPP←PostIOCB;           * LABEL done, DO DATA
        GOTO[EndSector];

* prepare for the label field
StartLabel: PFETCH1[DCBADDRESS, KData, IOCBLabelPointer];
        T←LabelSize, CALL[SetFieldConstants];
        RPP←LabelFlag, GOTO[NextField];                  * flag for label

* prepare for the data field
StartData: PFETCH1[DCBADDRESS, KData, IOCBDataPointer];
        T←DataSize, CALL[SetFieldConstants];
NextField: COMMANDWORD←LSH[COMMANDWORD, 2], GOTO[ProcessField];

* clean up after a data transfer
* ddwz+4 is the sector wakeup point after a data transfer
*** warning must reset data task allow before tasking
EndSector:   WR←WakeupAllow;
        OUTPUT[WR, DSKCTRLB];  * turn off data task allow
        WR ← WR, call[cReturn];  * interlock on WR insures complete before task switch
        nop;  * for the task, task must happen before this pfetch
        TASK, INPUT[DISKSTATUS, DSTATUS];  * status (after next sector mark)
        PFETCH1[DCBADDRESS, PROCESSWAKEUPREG1, IOCBTransferMask];   * NO ERROR INTERRUPT MASK TO PROCESSWAKEUPREG1
        DISKSTATUS←(DISKSTATUS) XOR (SectorMask);
        DISKSTATUS←(DISKSTATUS) OR (IOCBStatusValid);
        WR←ClearStatus, GOTO[DSWJL1, IOATTEN];
        OUTPUT[WR, DSKCTRLA];
        GOTO[SectorAll], RPP←PostIOCB;

DSWJL1: GOTO[DSWJ];

* set counts and pointers for field processing, call with field size in T
SetFieldConstants: LU ← KData1;
        ReadMemCount ← T, use CTask, GOTO[SetShortFieldConstants1, ALU=0];
        PFetch1[DCBADDRESS, KData1, IOCBAddressExtend];
        WriteMemCount ← T;  * avoid PFetch/pass around path problems
        LU ← (WriteMemCount) XOR (LabelSize);  * kludgy test for label or data
        GOTO[.+2, ALU=0];
          KData1 ← $DataExtend[KData1], GOTO[.+2];
          KData1 ← $LabelExtend[KData1];
        KData1 ← T ← LSH[KData1, 10];
        KData1 ← (RSH[KData1, 10]) + 1;
        use CTask;
        KData1 ← (LDF[KData1, 10, 10]) OR (T), return;

SetShortFieldConstants: ReadMemCount ← T, use CTask;
SetShortFieldConstants1: WriteMemCount ← T, return;

* task switch only for non-emulator wakeups
IOTask: LU←APCTASK;
        use CTASK,GOTO[cRETURN, ALU=0];
        NOP;
cRETURN: RETURN;

* errors come here
SeekFailure: NOP, GOTO[DSWJ];  * error entry points to DSWJ, error handler
DSWJ6: NOP, GOTO[DSWJ];
```

```
DSWJ7: NOP, GOTO[DSWJ];
DSWJ10:NOP, GOTO[DSWJ];
DSWJ:   INPUT[DISKSTATUS, DSTATUS];  * get disk status
        DISKSTATUS←(DISKSTATUS) XOR (SectorMask);  * invert sector info
        DISKSTATUS←(DISKSTATUS) OR (IOCBStatusValid);  * install 17 for alto
        DISKSTATUS←(DISKSTATUS) OR (IOCBHardwareError);
*        errcnt ← (errcnt) + 1 ;
        WR←WakeupAllow;  ***** warning this has to be done before tasking
        OUTPUT[WR, DSKCTRLB];  * clear data task allow, it might be set

* GET ERROR MASK, TASK BUT DONT CHECK ATTEN
* the current status will be posted into kblock and the dcb
DSWM:   RPP←ones; * SET ERROR FLAG into state control
        nop;*two m-i after output
        PFETCH1[DCBADDRESS, PROCESSWAKEUPREG1, IOCBErrorMask],call[cReturn];
*        NOP; * allow task switch before preceeding storage reference interlocks
        LU←DCBADDRESS;
        GOTO[DSWK, ALU#0], WR←ClearStatus;
          PROCESSWAKEUPREG1←0C; * clear my interrupt regester
DSWK:   OUTPUT[WR, DSKCTRLA], GOTO[SectorAll];   * DO A CLEAR STATUS

* IOCB command not valid
InvalidSeal:  DISKSTATUS←(DISKSTATUS) OR (IOCBCheckError), GOTO[DSWK];

* illegal sector
DSWL:DISKSTATUS←(DISKSTATUS) OR (IOCBIllegalSector), GOTO[DSWM];

DSWJL12:GOTO[DSWJ];  * error entry points
DSWJL:  GOTO[DSWJ];

        end[dm];
```

```
TITLE[EtherDefs];        * Defs for D0 microcode emulating Alto Ethernet

*Last modified by Murray on September 27, 1979  4:03 AM, Add EIdleTimer, EIReset
* modified by Johnsson on February 16, 1979  4:01 PM

        SET TASK [0]; *For R addressing

MC[EOReset,OR@[LSHIFT[EOTask,4],0]];
MC[EIReset,OR@[LSHIFT[EITask,4],0]];


* Ethernet I/O Address Registers
Set[EIData, 3];                    * Input data
Set[EIHost, 1];                    * Input data
Set[EStatus, 2];                   * Status/State register (read)
Set[EOData, 1];            * Output data
Set[EReadState, 2];  MC[ERState, 2];     * State register read
Set[EWriteState, 0];  MC[EWState, 0];    * State register write

* State Register command words
MC[ESetPurgeMode, 260];            * Enables input
MC[ESetOutputEOP, 107];            * Enables output, Jam
MC[EEnableInput, 220];
MC[EEnableOutput, 103];            * Enables Jam
MC[EDisableInput, 200];
MC[EDisableOutput, 100];                   * Clears OutputEOP, disables Jam
MC[EDisableInputOutput, 300];     * Disables input, output, clears outputEOP, Jam


* Status bits
SET [ESICOLL, 200];        * Receiver-detected collision (Jam)
SET [ESODL,  100];         * Output data late (Underrun)
SET [ESIDL,   40];                 * Input data late (Overrun)
SET [ESOCOLL,  20];        * Transmitter-detected collision (Collision)
SET [ESCRC,   10];                 * Bad CRC
SET [ESOFAULT,  4];        * Output DataFault (masked for now)
SET [ESOPAR,   2];                 * Output Bad Parity (masked for now)
SET [ESICMD,   4];                 * Input command issued  ** Not in hardware:
SET [ESOCMD,   2];         * Output command issued  **   for Alto emulation only
SET [ESIT,   1];           * Incorrectly terminated packet (Bad Alignment)

MC[EISMASK, ESIDL, ESCRC, ESIT];          * Status bits reported for input command
MC[EOSMASK, ESODL, ESOCOLL];     * Status bits reported for output command
MC[ECMDBITS, ESICMD, ESOCMD];    * Command bits


*  R-registers for input and output task

RV[EBase,0];     * Base register for first 64K space; 0 and 1 used by DoInt
RV[EBaseHi, 1];
RV[EPIR, 2];     * Buffer base register
RV[EPIRHi, 3];
RV[ECount, 4];   * Main loop counter
RV[ETEMP, 5];    * Temporary registers
RV[ETEMP1, 6];
RV[ETEMP2, 7];
RV[EFlag, AND@[OR@[LSHIFT[EITask,4],10],77]];    * input under output flag (reg 10 of EITask)
MC[pERandomReg, 377];              * Number of random number (REFR register)
MC[pEONotifyReg, 340];   * Register used for timer notify

*Dispatch table locations
Set[EEBase, LSHIFT[EEPage, 10]];
Set[EESIOLoc, ADD [EEBase, 20]];          * Dispatch location for SIO

*Address constants
Set[EOStartLoc, ADD[LSHIFT [EOPage, 10], 120]]; * Output notify location
Set[EOTimerDoneLoc, ADD[LSHIFT [EOPage, 10], 130]];     * Output TimerDone notify location
Set[EIStartLoc, ADD[LSHIFT [EIPage, 10], 140]]; * Input notify location
Set[EIAbortLoc, ADD[LSHIFT [EIPage, 10], 150]]; * SIO abort notify location

* Control block addresses (for Alto emulation, relative to 600)
MC[EPLOC, 0];    * Post location
MC[EPLOC1, 200];         * Post location (relative to 400)
MC[EBLOC, 1];    * Interrupt bit mask
MC[EBLOC1, 201];         * Interrupt bit mask (relative to 400)
MC[EELOC, 2];    * Ending word count
MC[ELLOC, 3];    * Load mask
MC[EICLOC, 4];   * Input count
MC[EICLOC1, 204];        * Input count (relative to 400)
MC[EIPLOC, 5];   * Input pointer
MC[EOCLOC, 6];   * Output count
MC[EOPLOC, 7];   * Output pointer
MC[EHLOC, 10];   * Host address for address recognition
MC[EHLOC1, 210];         * Host address for address recognition (relative to 400)

* Timer masks (slot is EOTask)
Set [ETimerRunning, 5];  * State 5 is simple timer
Set [ETimerIdle, 4];  * State 4 is idle
MC[ETimerMask, LSHIFT [ETimerRunning, 14]];
MC[EIdleTimer, LSHIFT [ETimerIdle, 14]];

*       ether constants required in both init and code (Midas mesa only)

* Microcode post codes (small integer in left half, ones in right half for XOR).
*  Note: value is complemented to get constant less than 8 bits.  Use XNOR for formation of post code.
MC[ESIDON,  NOT@ [377]];           * Input done
MC[ESODON, NOT@ [777]]; * Output done
MC[ESIFUL,  NOT@ [1377]];          * Input buffer overflowed
MC[ESLOAD, NOT@ [1777]];           * Load overflow
MC[ESCZER, NOT@ [2377]];           * Word count zero in input or output command
```

```
MC[ESABRT, NOT@ [2777]];        * Command aborted (by SIO)

* Miscellaneous
MC[ECOLLMASK, 10000];   * Mask for collision detection
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[EtherDefs];

        title[EtherInit];

*Last modified by Murray on September 12, 1979  7:45 PM  Add Dummy Output task Init
*Last modified by Johnsson on April 7, 1979  12:32 PM

        SET TASK [0]; *For register addressing.  This code really runs at task level EITask

*  ETHERNET INITIALIZATION subroutine (executed by EITask).
*  Will only be called if there is an Ethernet board in the machine.
*  Read Ethernet ID from board and form constant to be returned by SIO.

        ON PAGE [EtherInitPage];

EtherInit:      T ← GETRSPEC[103] xor (377C), at[EtherInInitLoc]; *Stkp (inverted)
        EBase ← pEHostRegx; *EHostReg is in the emulator's R space
        Stkp ← EBase;
        Input[Stack,EIHost];
        Stack ← (Stack) AND (377C);     * ID in right half
        Stack ← (Stack) OR (77400C);
*  Compute value for EONotify register, used for notify after timer wakeup.
        EBase ← AND@[0377, EOTimerDoneLoc]C;     * Low 8 bits of APC
        EBase ← (EBase) OR (OR@[lshift[EOTask,14],AND@[7400,EOTimerDoneLoc]]C); * High 4 bits of APC

        EBaseHi ← pEONotifyReg;
        Stkp ← EBaseHi, EBaseHi ← T;
        T ← EBase;
        Stack ← T;
        RETURN, Stkp ← EBaseHi; *restore Stkp
*Note - base registers are initialized each time input or output is started, so we
*do not need to do it here.

        RETURN, at[EtherOutInitLoc]; *Dummy Init for Out Task.

        end[etherinit];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[EtherDefs];
        title[EtherTask];

*Last modified by Murray on September 27, 1979  6:21 AM, Update for split state register
* modified by Murray on September 19, 1979  3:48 PM, Turn off Timer
* modified by Chang on May 31, 1979  1:09 PM, MIDAS Overlay
* modified by Chang on May 20, 1979  2:01 PM, nail down EESIO
* modified by BRD on March 23, 1979  10:34 PM
*       added RS232 SIO code dispatch
* Modified by Johnsson on April 7, 1979  12:33 PM

        SET TASK [0];

* EMULATOR TASK -- SIO instruction

        ON PAGE [EEPage];
*This code executes at task 0
* We get here after the SIO instruction has been issued.
*The SIO control bits are in T (bits 16,17).
* Get host address constant for Ethernet
*RTEMP1 = 1 if called from Mesa, 0 if called from Nova
*This code is really part of the emulator, and uses its temporary registers
EESIO:  AC0 ← T, at[EEStartLoc]; *save control bits (useful only if called from Mesa)
        T ← LDF[AC0,10,2];
        RTemp ← AND@[377, RS232SIOLoc]C;
        RTemp ← (RTemp) OR (OR@[LSHIFT[16,14],AND@[007400, RS232SIOLoc]]C);
        RTemp ← (RTemp) OR (T);
        APC&APCTask ← RTemp, TASK;
        Return;
        T ← 177C;
        T ← (ldf[EHostReg,0,10]) xor (T); *these bits will be 177b if the init code was run
*(i.e. if there is an Ethernet board in the machine), and will be zero otherwise
        T ← EHostReg, goto[EESIODisp,ALU=0];
        AC0 ← 0C;
        AC0 ← (AC0) xnor (100000C),goto[EESIO0]; *return 77777b in AC0 if no Ethernet board

* Dispatch on low 2 bits of AC0
EESIODisp:      Dispatch[AC0,16,2];
        AC0 ← T, DISP [EESIO0]; * Host Address
* 00 -- Do nothing
EESIO0: lu ← RTEMP1, dblgoto[EEMesaRet,EENovaRet,Rodd], AT[EESIOLoc, 0];

EENovaRet:      loadpage[nePage];
        FF1@[17], gotop[neNoskip];

EEMesaRet:      loadpage[7];
        gotop[P7Tail];


* 01 -- Start transmitter
* Form APC&APCTask word to notify the output microcode
EESIO1: RTEMP ← AND@[0377, EOStartLoc]C, AT[EESIOLoc, 1];        * Low 8 bits of APC
        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EOTask,14],AND@[007400, EOStartLoc]]C);
* 10 -- Start receiver
* Form APC&APCTask word to notify the input microcode
EESIO2: RTEMP ← AND@[0377, EIStartLoc]C, AT[EESIOLoc, 2];        * Low 8 bits of APC
EESIO2A:        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EITask,14],AND@ [007400, EIStartLoc]]C);
* 11 -- Reset interface, i.e. abort.  Input task is notified to post abort.
EESIO3: GOTO[EESIO2A], RTEMP ← AND@[0377, EIAbortLoc]C, AT[EESIOLoc, 3];        * Low 8 bits of APC

* Notify appropriate code
EESIONotify:
        RCNT ← AND@[EOTask,17];
        RCNT ← (RCNT) + (EIdleTimer);
        LoadTimer[RCNT];
        T ← EOReset;
        RCNT ← (EDisableInputOutput);
        OUTPUT[RCNT];
        CALL[ETaskRet], APC&APCTASK ← RTEMP;
        lu ← RTEMP1, dblgoto[EEMesaRet,EENovaRet,Rodd]; *control returns to here when emulator next runs


*
*
* INPUT TASK MICROCODE
*

        ON PAGE [EIPage];
* Input microcode is notified at EIStart by the emulator (at SIO).
* Some initialization is done, and the TPC set up to EIIDLE.
* Initialize, enable hardware, and TASK
EIStart:        CALL [EINIT], ETemp ← EEnableInput, AT [EIStartLoc];
* Wake up here when first word of a new packet arrives.
* Address filtering.
EIIDLE: INPUT [ETemp2, EIData];         * Read  in first word
        T ← RSH [ETemp2, 10];           * Right justify destination host in T
        SKIP [ALU#0];                   * Check for broadcast
        GOTO [EIBegin], EFlag ← 1C;             * Broadcast packet
        PFetch1 [EBase, ETemp1, EHLOC!];        * Fetch host address
        LU ← (ETemp1) XOR (T);
        SKIP [ALU#0], LU ← ETemp1;      * ALU=0 => destination host = me
        GOTO [EIBegin], EFlag ← 1C;     * Packet for me
        GOTO [EIPurge, ALU#0];          * ALU=0 => host is promiscuous
        GOTO [EIBegin], EFlag ← 1C;     * Packet for me

* Packet not accepted by filter, so tell hardware to ignore the rest of this packet.
* Purge packet, and TASK
EIPurge:        ETemp ← ESetPurgeMode;
```

```
	OUTPUT [ETemp, EWriteState];
	CALL[ETaskRet];
* Wakeup here at start of next packet
	GOTO [EIIDLE];

* Packet accepted by filter.
* EFLAG is set to 1 to tell the output microcode that a packet came in
*  (used for input under output).
* Disable output task if it is on (probably in retransmission wait).
EIBegin:	T ← EOReset;
	ETemp1 ← EDisableOutput;
*	OUTPUT [ETemp1];  Oops, kills us, not him.
	CALL[ETaskRet];
* Set up EPtr and ECount for single word transfers.
	CALL[EBSetup], T ← EICLOC;		* Set up EIPTR, ECount
* Subroutine returns with:  EPtr = IPtr + ICount - 1,   ECount = - ICount
* Check if buffer count zero.  If not first word gets stored in EIAlign loop.
	GOTO [EICountZero, R>=0], T ← ECount ← (ECount) + 1;	* R>=0 => count is zero
* Check buffer alignment.
* Compute how many singles before first quadword, and form loop counter in ETemp1.
*	Address:  x00 => no singles, loop count = -1
*	Address:  x01 => 3 singles, loop count = 2
*	Address:  x10 => 2 singles, loop count = 1
*	Address:  x11 => 1 singles, loop count = 0
	T ← (EPtr) + (T) + 1;		* Form start address in T
	ETemp1 ← (ZERO) - (T);		* Complement, increment
	ETemp1 ← (LDF[ETemp1, 16, 2]) - 1;
	CALL [EIAlignE], T ← ECount;		* Set return for EIAlign loop
EIAlign:	GOTO [EIQuad, R<0], ETemp1 ← (ETemp1) - 1;
	GOTO [EIBufFull1, R>=0], T ← ECount ← (ECount) + 1;
	INPUT [ETemp2, EIData];
	LU ← ETemp2;		* Abort
EIAlignE:	RETURN, PStore1 [EPtr, ETemp2];
* Now start quadword output.
* Adjust EPtr and ECount for 4-word transfers.
EIQuad: ECount ← (ECount) + (3C);
	CALL [EILoop], EPtr ← (EPtr) - (6C);	* Set return address for EILoop
* Read the Hardware Input Buffer into the Main Memory In Buffer.
EILoop: GOTO [EIQuadFull, R>=0], T ← ECount ← (ECount) + (4C);
	GOTO[EIAttn, IOATTEN];
	RETURN, IOStore4[EPtr, EIData];
* Get here when no more room for quadwords in buffer.
* Check IOATTEN is high to see if end of packet.
EIQuadFull:	GOTO [EIAttn1, IOATTEN];
* Not end of packet.  Do singles to fill buffer.
* 7-ECount = number of singles remaining in buffer.
* Set up loop counter as (- No. singles), and read in singles.
	ECount ← (ECOunt) - (7C);
	CALL [EISingles];
EISingles:	GOTO [EIBufFull, R>=0], ECount ← (ECount) + 1;
	GOTO [EIAttnS, IOATTEN], T ← (ECOunt) + (6C);	* Set up T for PStore1
	INPUT [ETemp, EIData];
	LU ← ETemp;
	RETURN, PStore1 [EPtr, ETemp];
* We get here when IOATTEN is detected in EILoop.
* Number of word left in buffer = 7 - ECount + 1 (CRC) + Excess count.
EIAttn1:	NOP;
EIAttn: INPUT [ETemp, EStatus];		* Read Status
	T ← LDF[ETemp, 10,2];		* Isolate Excess Count
	ECount ← (ECount) XNOR (0C);	* Complement
	ECount ← (ECount) + (11C);	* Increment, add 8
	ECount ← (ECount) + (T);		* Add excess count
EIAttn2:
*	T ← ETemp;
*	ETemp ← (ETemp) AND (124400C);
*	SKIP [ALU=0];
*	NOP%BREAKPOINT%;
*	ETemp ← T;
	ETemp ← (RSH[ETemp, 10]);		* Shift down status
	ETemp ← (ETemp) AND (EISMASK);	* Mask out uninteresting status bits
	ETemp ← (ETemp)XNOR(ESIDON);	* Post input done status
* Store EECLOC.
EIPost: CALL[ETaskRet], PStore1 [EBase, ECount, EELOC!];
* Post status, disable interface (purge packet too), and TASK.
* Post status in ETemp, disable value in ECount.
EIPostA:	ECount ← EDisableInput, CALL [EPost];
*  End of packet.
	EBaseHi ← 0C; *repair base registes smashed by EPost (DoInt)
	EBase ← 200C;
	EBase ← (EBase) or (400C), GOTO [EIIDLE];


* Get here when an IOATTEN is detected during the EISingles loop.
* Number of words left in buffer = 1 - ECount + 1 (CRC).
EIAttnS:	ECount ← (ECount) XNOR (0C);	* Complement
	ECount ← (ECount) + (3C);		* Increment, add 2
	GOTO [EIAttn2], INPUT [ETemp, EStatus];
* We get here when the input buffer is exactly full.
* First check if IOATTEN is high, indicating that the last word was the CRC.
EIBufFull1:	NOP;
EIBufFull:	SKIP[NOATTEN], ETemp ← 1C;
	GOTO [EIAttn2], INPUT [ETemp, EStatus]; *Last word input was CRC
* Read one more word to see if the next is the CRC word (which we will discard).
	ECount ← 0C;		* No words left in buffer
	CALL[ETaskRet], INPUT [ETemp, EIData];
* After wakeup, check IOATTEN.
	NOP;		* Can't check for Attn here
	GOTO [EIAttn2, IOATTEN], INPUT [ETemp, EStatus];	* IOATTEN => Word was CRC
	ETemp ← 0C;
	GOTO [EIPOST], ETemp ← (ETemp) XNOR (ESIFUL);	* Input buffer overrun, post status
```

```
* Get here if input buffer has zero word count.  Post.
EICOuntZero:    ETemp ← 0C;
        GOTO [EIPost], ETemp ← (ETemp) XNOR (ESCZER);


* Input microcode is notified here by emulator SIO when AC0[16:17] = 3.
* Manufacture "Abort" status and post. (Input hardware will be disabled again, which doesn't matter.)
EIAbort:        EBaseHi ← 0C, AT [EIAbortLoc];  * Set up base pointer to loc 600
        EBase ← 200C;
        EBase ← (EBase) OR (400C);
        ETemp ← ECMDBITS;
        GOTO [EIPostA], ETemp ← (ETemp) XNOR (ESABRT);
*
*
* Alto-emulation OUTPUT MICROCODE
*

* Output microcode is notified at EOStart by the emulator (at SIO).
* Some initialization is done, and the TPC set up to EOIDLE, enable hardware and TASK.
EOStart:        CALL [EINIT], ETemp ← EEnableOutput, AT [EOStartLoc];

* Idle state of the Ethernet output task

 EOIDLE:        PFetch1 [EBase, ETemp1, ELLOC!];        * Fetch current load
        SKIP [R>=0], T ← (LSH[ETemp1 , 1]) + 1; *  Form new load, check if old overflowed
        GOTO [EOLDOV], ETemp ← 0C;      * Post Load overflow status
        ETemp ← T;                      * Store updated load in ELLOC
        T ← ELLOC, TASK;
        PStore1 [EBase, ETemp];         * Store new load
* Compute countdown interval
*  Get random number from "random" register (REFR register used).
        T ← Stkp;                       * Save Stkp and
        ETemp2 ← pERandomReg;   *  point to "random" register
        Stkp ← ETemp2, ETemp2 ← T;
        T ← LDF [Stack, 4, 10]; * Get bits 4-13
        ETemp2 ← (ETemp2) XOR (377C);   * Complement Stkp value
        Stkp ← ETemp2;          *  and restore
        ETemp1 ← (ETemp1) AND (T);      * Mask random number (ETemp1 has Load mask)
        GOTO [EOSetup, ALU=0], ETemp1 ← LSH[ETemp1,1];  * Scale for 5.44 us ticks
        T ← (LDF[ETemp1, 7, 2]) - 1;
        ECount ← T, TASK;               * Save high part (minus 1) (2 bits)
        EFlag ← 0C;     * Clear Input under Output Flag
        ETemp1 ← LDF[ETemp1, 11, 7];    * ETemp1 now has low 7 bits of random number
* Before starting timer, check if input is set up.
* If the input word count is nonzero, enable the receiver while waiting to transmit.
        PFetch1 [EBase, ETemp, EICLOC!];
        LU ← ETemp;
        T ← EIReset;
        SKIP [ALU=0], ETemp ← EEnableInput;
        OUTPUT[ETemp];
        ETemp ← EDisableOutput; * ALU=0 => no input set up
        OUTPUT [ETemp, EWriteState];
* Start simple timer with low 7 bits of random number.
* Timer slot is EOTask.
        ETemp2 ← ETimerMask;    * Compute timer word
        T ← CTASK;      * Timer slot is same as output task no.
        T ← (ETemp2) OR (T);
EOLoadTimer:    ETemp1 ← LSH[ETemp1, 4];
        ETemp1 ← (ETemp1) OR (T), TASK;
        LoadTimer[ETemp1];

*  Timer has expired.  Check if input (under output) came in.
EOTimerDone:    GOTO [EOMoreTime, R EVEN], LU ← EFlag, AT [EOTimerDoneLoc];     * Check if pkt came in (EFlag = 1)
        ETemp ← EDisableOutput;         * If so, abort output
        OUTPUT [ETemp, EWriteState];
        CALL[ETaskRet];
        NOP;
* Check if still more time to elapse before start of transmission (High part of random number >=0).
EOMoreTime:     ETemp1 ← 177C;          * Set up maximum timer value
        GOTO [EOLoadTimer, R>=0], ECount ← (ECount) - 1;
* Enable output and shut off the receiver (in case it was turned on).
EOBegin:        T ← EIReset;
        ETemp ← EDisableInput;
        OUTPUT[ETemp];
        ETemp ← (EEnableOutput);
        OUTPUT [ETemp, EWriteState];
* Set up EPtr and ECount for single word transfers.
EOSetup:        T ← EOCLOC;
        CALL[EBSetup];
* Subroutine returns with:  EPtr = OPtr + OCount - 1,  ECount = -OCount
* Check for zero count.
        GOTO [EOCountZero, R>=0], LU ← ECount;  * R<0  => count is zero
* Compute how many singles before first quadword, and form loop counter in ETemp1.
*       Address:  x00 => no singles, loop count = -1
*       Address:  x01 => 3 singles, loop count = 0
*       Address:  x10 => 2 singles, loop count = 1
*       Address:  x11 => 1 singles, loop count = 2
        T ← ECount;
        T ← (EPtr) + (T) + 1;   * Form start address in T
        ETemp1 ← (ZERO) - (T);  * Complement, increment
        CALL [EOAlign], ETemp1 ← (LDF[ETemp1, 16, 2]) - 1;
EOAlign:        GOTO [EOQuad, R<0], ETemp1 ← (ETemp1) - 1;
        GOTO [EONoMore1, R>=0], T ← ECount ← (ECount) + 1;
        PFetch1 [EPtr, ETemp];
        LU ← ETemp;             * Abort
        GOTO[ERet], OUTPUT [ETemp, EOData];
* Now start quadword output.
* Adjust EPtr and ECount for 4-word transfers.
EOQuad: ECount ← (ECount) + (3C);
        CALL [EOLoop], EPtr ← (EPtr) - (6C);    * Set return address for out loop
* Output from the Main Memory Output Buffer to the Hardware Output Buffer.
```

```
EOLoop: GOTO [EOQuadEmpty, R>=0], T ← ECount ← (ECount) + (4C);
        GOTO [EOAbort, IOATTEN];
        RETURN, IOFetch4 [EPtr, EOData];

* Normal exit from Output Loop is here
* 7 - XWCount = number of singles remaining
* T is set up for next location.
EOQuadEmpty:    ECount ← (ECount) XOR (7C);
        CALL[EOSingles], ECount ← (ECount) - 1;
EOSingles:      GOTO [EONoMore, R<0], ECount ← (ECount) - 1;
        PFetch1 [EPtr, ETemp];
        T ← (ZERO) + (T) + 1, ETemp;    * Abort
        GOTO[ERet], OUTPUT [ETemp, EOData];
* We're done outputing words. Set output EOP.
EONoMore1:      NOP;
EONoMore:       ETemp ← ESetOutputEOP;
        OUTPUT[ETemp,EWriteState];      * Set OutputEOP
        CALL[ETaskRet];
* Should be woken up here after hardware's done sending packet or an error
EOEND:  INPUT [ETemp, EStatus];         * Read Status
EOEND1:
*       T ← ETemp;
*       ETemp ← (ETemp) AND (53000C);
*       SKIP [ALU=0];
*       NOP %BREAKPOINT%;               * BREAK - Bad Ouput Status
*       ETemp ← T;
        LU ← (ETemp) AND (ECOLLMASK);   * Look at collision bit
        GOTO [EOCOLL, ALU#0], ETemp←EDisableOutput;     * ALU#0 => Collision, try again
* If not collision, form status. Could be good packet or underrun (ODL).
        ECount ← 0C;
        ETemp ← RSH[ETemp, 10];         * Shift down status
        ETemp ← (ETemp) AND (EOSMASK);  * Remove uninteresting bits
        ETemp ← (ETemp) XNOR (ESODON);
EOPost: CALL[ETaskRet], Pstore1 [EBase, ECount, EELOC!];        * Store end count
        ECount ← EDisableOutput, CALL [EPost];
        GOTO[.]%BREAKPOINT%;            * Shouldn't get here.
*  We arrive here after an IOATTEN is detected in the main loop, indicating an error condition.
* IOATTEN will be true if a collision or underrun has occurred.
EOAbort:        GOTO [EOEND1], INPUT [ETemp, EStatus];  * Now read status
*
* Collision encountered, disable hardware to clear collision, enable and try again.
EOCOLL: OUTPUT[ETemp, EWriteState];
        ETemp1 ← EEnableOutput;
        OUTPUT[ETemp1, EWriteState];
        CALL[ETaskRet];
        GOTO [EOIDLE];
* Load overflow, post status (NOT [ESLOAD], ETemp is 0)
EOLDOV: GOTO [EOPost], ETemp ← (ETemp) XNOR (ESLOAD);
* Output buffer count is zero. Post (NOT [ESCZR]).
EOCountZero:    ETemp ← 0C;
        GOTO [EOPost], ETemp ← (ETemp) XNOR (ESCZER);
*
*
*
*
* Task-independent Subroutines


*  Subroutine [EPost];
* Posts the command completion, and wakes up driver.
* Expects post code and status in ETemp.
* ECount has disable code to be sent to State register.
EPost:  PFetch1 [EBase, ETemp2, EBLOC!];        * Fetch wakeup mask;
        PStore1 [EBase, ETemp, EPLOC!]; * Store ending status in EPLOC
* Now wakeup driver.
        NOP;                            * wait for write of ECount
        OUTPUT [ECount, EWriteState];
        LoadPage[0];
        T ← (ETemp2) or (100000c), gotop[DoInt];        * Get wakeup mask from ETemp. 100000c means tasking return

* Subroutine [EBSetup].
* Set up EPtr and ECount register.
* On entry T has pointer to EICLOC or EOCLOC.
* Subroutine returns with:
*       EPtr = Buffer Pointer + Count - 1
*       ECount = - Count
* The appropriate input or output pointer and count locations are used.
* Subroutine knows that EIPLOC = EICLOC + 1, and EOPLOC = EOCLOC + 1.
EBSETUP:        PFetch1 [EBase, ECount];                * Fetch count
        NOP;                            * So T can be written
        T ← (ZERO) + (T) + 1;           * Point to ExPLOC
        PFetch1 [EBase, EPtr];          * Fetch pointer
        T ← (ECount) - 1;
        EPtr ← (EPtr) + (T);            * Ptr ← Ptr + count - 1
        RETURN, ECount ← (ZERO) - (T) - 1;      * Count ← - Count

* Subroutine [EINIT].
* Initialization subroutine.
* Called by both input and output task.
* ETemp contains the enable code to be used to enable the hardware.
EINIT:  EBaseHi ← 0C;           * Set up base pointer to loc 600
        EBase ← 200C;
        EBase ← (EBase) OR (400C);
        EPtrHi ← 0C;                    * Set up high part of Buffer pointer
        GOTO[ERet], OUTPUT [ETemp, EWriteState];


* These instructions used for Tasking after OUTPUT instructions.
ERet:   NOP;
ETaskRet:       RETURN;
```

```
end[ether];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;
insert[GlobalDefs]; *task and page assignments

TITLE[FaultHandler];

*last edit by Johnsson, October 15, 1979  10:44 AM, AR 2369
* edit by Kennedy, October 13, 1979  2:57 PM, Allow 16-bit fault codes
* edit by Kennedy, October 3, 1979  9:15 AM, Update MP Codes
* edit by Chang, August 22, 1979  5:01 PM, MOBCrash=212B/138D, fix LogSE
* edit by Johnsson, June 27, 1979  8:21 AM, Log MC2 errors if LogSE
* edit by Johnsson, June 18, 1979  5:19 PM, new Midas
* edit by Chang, May 21, 1979  2:33 PM, nail down StartMemTrap
* edit by Johnsson, May 10, 1979  9:42 AM

* split off from Initialize by Johnsson, April 4, 1979

*Error Codes for Fault Handler
MC[CrashOffset,1000]; * Offset for upper 8-bits of fault codes.
MC[RCSCrash,330]; * 728d - R or CS parity error
MC[BPCrash,331]; * 729d - Real Breaakpoint
MC[MOBCrash,342]; * 738d - Map Out of Bounds
MC[H4PECrash,333]; * 731d - H4PE
MC[MOB&H4PECrash,334]; * 732d - MOB and H4PE
MC[MC2Crash,335]; * 733d - got an MC2 error when unable to handle it by RETURNing
MC[MC22Crash,336]; * 734d - 2 MC2 errors
MC[MC1Crash,337]; * 735d - MC1 fault when emulator couldn't accept it
MC[LPCrash,340]; * 736d - Fault from the instruction following a LoadPage
MC[StkCrash,341]; * 737d - Stack over/underflow
  MC[NStkCrash,36]; * Complement of StkCrash (8 bits)

*Fault handler
*This code starts at location 120, task 17.
*Control gets here from the kernel as soon as the state has been saved.
*This code determines what to do based on the type of error, and the bits in FFAULT.
*Bits in FFAULT are:
*        0: MC2 errors RETURN if 1, crash if 0
*        1: Midas is present (1), so "crash" means breakpoint, else put a
*           code in MP and halt.
*       15: MC1/StackOvf errors handled by notifying PFEntry in emulator (1),
*           or by crashing (0)

* breakpoint means notify task 17, location 7514 (Kernel location BPSEND).


        OnPage[0];
        SetTask[17];

RV[PipeReg,60]; *Pipe Ram Entry goes here
        MC[pPipeReg,360];
        MC[pPipeReg2,362];
RV[PipeReg4,64];
RV[PipeReg5,65];

SET[H4Disp,240];
SET[Mc2ErDisp,260];

FaultStart: lu ← (RXPPB) and (3000c), at[120]; *test R & CS parity
        goto[RCSErr, ALU#0], lu ← (RXPPB) and (400c);   *test memory error
        goto[MC12Err, ALU#0], lu ← (RXPPB) and (4000c); *test stack ovf
        goto[TryBP, ALU=0];
StkEr:  FFAULT, dblgoto[MC1NotifyEmulator, Crash,RODD], T ← StkCrash; *can emulator take fault?
RCSErr: T ← RCSCrash, goto[Crash];
TryBP:  T ← BPCrash, goto[Crash];

*Get here with error code in T.  If Midas is present, breakpoint.  Otherwise, put
*the code into the maintenance panel and halt.
Crash:  lu ← ldf[FFAULT,1,1];
        goto[Midas,ALU#0],PipeReg5 ← (lsh[PipeReg5,10]) or (T); *save error code in right half of PipeReg5
        RTMP ← CrashOffset;
        T ← (RTMP) + (T), call[PNIP]; *add offset to fault code and then display it
        goto[.];

Midas:  lu ← ldf[RXPPB,4,4]; * test parity register
        RTMP ← 177400c, skip[alu#0]; *Notify Midas at 7510b or 7512b
          RTMP ← (RTMP) OR (110C), goto[MidasNotify]; *Go overlay 'Break'
        RTMP ← (RTMP) OR (112C); *Go overlay 'MidasFault'
MidasNotify:
        APC&APCTASK ← RTMP;
        RETURN;


MC12Err:
        Stkp ← RXSTK; * not stack error, restore pointer
        ReadPipe[PipeReg];        *get A pipe
        Dispatch[PipeReg,4,2];  *dispatch on H4pe, MapBnd
        Dispatch[PipeReg,0,2], Disp[NoH4BndEr]; *dispatch on MC2ErA', MC2ErB'

NoH4BndEr:      Disp[MC2ErAB], lu ← (PipeReg) and (20000c), AT[H4disp,0]; *test MC1ErA' bit
BndEr:  T ← MOBCrash, goto[Crash], AT[H4disp,1]; *MOB error only
H4Er:   T ← H4PECrash, goto[Crash], AT[H4disp,2]; *H4PE only
H4BndEr:        T ← MOB&H4PECrash, goto[Crash], AT[H4Disp,3]; *H4PE & MOB

MC2ErAB:        T ← MC22Crash, goto[Crash], AT[MC2ErDisp,0]; *Have both MC2 A & B error - crash
MC2ErA: T ← lhmask[MemSyndrome], goto[MC2Er], AT[MC2ErDisp,1]; *MC2A error
MC2ErB: ReadPipe[PipeReg], ResetMemErrs, AT[MC2ErDisp,2]; *MC2B error - read pipe entry
        T ← lsh[MemSyndrome,10], goto[MC2Er];
NoMC2Er:        Goto[MC1Er,ALU=0], ResetMemErrs, AT[MC2ErDisp,3]; *Branch if MC1ErA' = 0
        ReadPipe[PipeReg], ResetMemErrs; *MC1B error - read pipe entry
MC1Er:  FFAULT, dblgoto[MC1NotifyEmulator,Crash,RODD], T ← MC1Crash; *can emulator take fault?
```

```
* As an interim approximation to error logging, MC2 errors occuring
* on a page with LogSE set will store the pipe data + symdrome at
* VM 710 and resume.
MC2Er:  PipeReg ← (rhmask[PipeReg]) or (T);
*       ResetMemErrs, T ← PipeReg5; *single error problem
        ResetMemErrs, FFault, goto[MC2ErRet, r<0];
        LU ← LDF[PipeReg5,10,1]; * getting the flag bit
*       RTMP ← 377c, goto[MC2Log, NoH2Bit8]; * test (inverted) LogSE
        RTMP ← 377c, goto[MC2Log, alu=0]; * test (inverted) LogSE
        T ← MC2Crash, goto[Crash];
MC2ErRet:
        return;

* RTMP ← 377c will be used as odd base register.  Since there is
* no overflow and RTMP[0:7]=0 all will work correctly.
MC2Log: T ← 311c;         * 311+377 = 710
        PStore4[RTMP, PipeReg];
        T ← 315c;         * 315+377 = 714
        PStore2[RTMP, PipeReg4], goto[T17Restore];

*Notify emulator at EmNotifyA if emulator was the interrupted task, else at EmNotifyB
MC1NotifyEmulator:
        lu ← ldf[RXCTask,0,4];
        RTMP ← 202c, goto[MC1NEx,ALU#0];
        RTMP ← 200c; *emulator task interrupted - notify EmNotifyA
        MNBR ← RXCTask; *save the emulator's PC for further consideration
MC1NEx: APC&APCTask ← RTMP;
        SALUF ← T, return; *save the crash code in saluf, so that the emulator can use it

T17Restore: RXCTask ← (RXCTask) xnor (170000c), AT[204]; *complement CIA
        T ← ldf[RXCTask,4,4];   *page bits
        lu ← (ldf[RXPPB,0,4]) xor (T);  *compare with saved page register
        goto[LoadPageError,ALU#0], T ← lsh[RXALU,4]; *result register
        APC&APCTask ← RXCTask;
        RETURN,RESTORE, A ← RXAPC, lu ← T; *back to faulted task

LoadPageError: T ← LPCrash, goto[Crash];


        SetTask[0];
NotifyBack: RTEMP ← (RTEMP) or (170000C);
        APC&APCTask ← RTEMP, goto[PFExit];
PFExit: return;

EmNotifyA:      usectask, xBuf ← T, AT[200]; *save the emulator's T in xBuf
        T ← APC&APCTask, call[PFExit]; *save emulator's TPC in T, and task switch

*       PF handling starts here if the emulator was interrupted.
*       Note that if the emulator was NOT interrupted, then the
*       fault cannot have come from buffer refill.  Since control
*       entered here, the emulator's PC (complemented) is in MNBR.
PFEntryA:
        xBuf1 ← T, loadpage[FaultPage1];
        gotop[PFEntryAx];

EmNotifyB:
        usectask, xBuf ← T, AT[202];
        T ← APC&APCTask;
        RTEMP ← 204c, call[NotifyBack]; *Prepare to notify back to task 17, location T17Restore

*       PF handling starts here if non-emulator was interrupted.
PFEntryB: xBuf1 ← T, loadpage[FaultPage1];
        gotop[PFEntryBx];


        OnPage[FaultPage1];

PFEntryAx:
        T ← NStkCrash, Call[CheckStackTrap];
        lu ← ldf[xBuf2,4,4], goto[CheckBufferRefill];

PFEntryBx:
        T ← NStkCrash, Call[CheckStackTrap];
        goto[CMSt1], Dispatch[MemStat,15,3]; *cannot be buffer refill trap


CheckStackTrap: lu ← (SALUF) xor (T);
        T ← MNBR, goto[StackErrorz, ALU=0]; *set up to test page bits of emulator's PC
        xBuf2 ← (ZERO) xnor (T); *complement value
        xBuf2 ← (xBuf2) and not (170000c), return;
*We have a stack error.  Cause the trap immediately. Set Stkp back to beginning of last
*instruction.  Let the PC fall where it may.

StackErrorz:
        lu ← (GetRSpec[103])-(lshift[11,10]c); *test SStkp for 9 or more
        T ← SStkp, skip[nocarry];
          RTEMP ← 10c, goto[.+2];
        RTEMP ← T;
        Stkp ← RTEMP;
        loadpage[7];
        T ← sStackError, gotop[kfcr];


*At this point, xBuf contains the emulator's T at the time of the fault,
*xBuf1 contains  the emulator's TPC, and xBuf2 contains the PC of the
*aborted instruction.  The major problem with faults is to determine the
*PC to store in the frame, and whether to continue the instruction, or
*cause the trap immediately.  The cases are:

*1) The fault was caused by a NextInst (PC is on page 0, TPC points to
```

```
*a NextInst). We set IBUF to -1, PCF to 0, and send control back to
*the NextInst. This will cause the faulted instruction to be completed,
*and control will go to opcode 377, which will cause the trap with
*PC = 2*PCB + PCX -1.

*2) The fault was due to a NextData in the first microinstruction of
*a bytecode (TPC = 01xxxxxxxx01). The trap is started immediately, with
*PC = 2*PCB - 1 (The NextData was trying to get an operand from location
*0 of the buffer, so the opcode is at location 7 of the previous buffer,
*but PCB was incremented by 8 bytes before the fault was discovered).

*3) The fault was detected on page 6 and was due to a PFetch4. This is
*a jump instruction buffer refill. We proceed as in case 1 without
*setting PCF.

*4) The fault was due to an Xfer buffer refill (MemStat[13:15] =
*XferFixup). This is handled just like a jump.

*5) The fault occured during the early phases of Xfer. We want to back
*out and redo the instruction, but CODE may have changed and we need it
*to compute the PC to save. Call Loadgc to reload from the current LOCAL.

*If none of these situations hold, the PC is (PCB*2) + q, where q = if
*(PCF>=PCX) then PCX-1 else PCX-9 (if PCF<PCX, then the buffer was
*refilled between the NextInst and the fault, and PCB has been advanced
*by 8 bytes). In the normal case, the trap is started using this PC, and
*it is not necessary to unwind the instruction. If any special unwinding
*is necessary,it is indicated by a value in MemStat[13:15].


CheckBufferRefill:
        T ← 6c, goto[CameFromPage0,ALU=0]; *test for emulator page 0 fault
        lu ← (ldf[xBuf2,4,4]) xor (T);
        T ← (GETRSPEC[103]) xor (377c), goto[CheckMemStat,ALU#0]; *get ready to save stackpointer
*We came from page 6. If the operation was PFetch4, this is a jump buffer refill
        IBuf3 ← pPipeReg2; *IBuf3 is a guaranteed free temporary
        Stkp ← IBuf3, IBuf3 ← T, task; *now pointing at the operation
        T ← 11c; *not 6 (PFetch4)
        lu ← (ldf[Stack,14,4]) xor (T);
        Stkp ← IBuf3, dblgoto[CMSt2,ContinueInterruptedBytecode,ALU#0];
*We have a PFetch4 from Page 6, i.e. JumpCity. We continue the jump after filling IBuf with -1's,
*and eventually get to opcode 377.


CameFromPage0:
        xBuf1 ← lcy[xBuf1,6]; *set up to test for TPC = 01xxxxxxxx01.
        lu ← xBuf2;      * test for aborted pc = 0
        T ← 101c, goto[FPCOy, alu = 0]; * high 2 and low 2 bits or addr, task 0 in middle
        lu ← (rhmask[xBuf1]) xor (T);
        goto[CBR1, ALU#0], T ← rcy[xBuf1,6]; * put TPC back
        PCB ← (PCB) -(4c); *TPC = 01xxxxxxxx01, fault was from first instruction of bytecode
        PCF ← AllOnes; * PCF ← 7
        T ← (GetRSpec[103]) xor (377c), goto[SMTrpx]; * use current Stkp

CBR1:   xBuf2 ← T;       * xBuf2 now contains TPC instead of aborted PC
        T ← 0c; *Does TPC point to a NextInst?
        APC&APCTask ← xBuf2;
        ReadCS;
        CSDATA, goto[CMSt0,Rodd];
        PCF ← RZERO;    *PCF ← 0
ContinueInterruptedBytecode:    T ← xBuf; *xBuf2 points to place to resume the bytecode
CIB1:   IBuf ← (Zero)-1, task;  *force bytecode 377
        IBuf1 ← (Zero)-1;
        IBuf2 ← (Zero)-1;
        IBuf3 ← (Zero)-1;
        APC&APCTask ← xBuf2, goto[PFExit1];     *Return to the NextInst


SET[FixDisp,ADD[LSHIFT[FaultPage1, 10], 100]];

CMSt2:  Dispatch[MemStat,15,3],  goto[CMSt1];
CMSt0:  Dispatch[MemStat,15,3],  goto[CMSt1];
CheckMemStat:   Dispatch[MemStat,15,3];
CMSt1:  Disp[FixPCOnly];


FixXfer:        T ← xBuf, goto[CIB1],AT[FixDisp,1];

FixEarlyXfer:   LoadPage[xfPage1],AT[FixDisp,4];
        PFetch1[LOCAL,xfTemp,0], callp[Loadgc];
FPCOy:  T ← (PCXReg) - 1, goto[FPCOx];



FixBLTL:        T ← SStkp,AT[FixDisp,2];        * prepare for fixup relative to saved stkp
        RTEMP ← (T), task;
        RTEMP ← (RTEMP) - (4c);
        Stkp ← RTEMP;
        T ← xBuf,Call[BumpGlorp];       * source + T
        Stack&+1;
        Stack ← (Stack)+1;              * count + 1
        Stack&+1, Call[BumpGlorp];      * dest + T
        T ← (PCXReg) - 1, goto[FPCOx];  * one byte inst cannot have refilled buffer

BumpGlorp:      Stack ← (Stack)+(T);
        Stack&+1,skip[NoCarry];
        Stack ← (Stack)+1,return;
PFExit1:
        return;
```

```
FixBlt: T ← (SStkp) - 1, AT[FixDisp,3]; * prepare for fixup relative to saved stkp
        RTEMP ← pPipeReg2, task;
        Stkp ← RTEMP, RTEMP ← T; *Stack points at operation, RTEMP points to count word
        * Operation (complemented) is low order four bits of Stack
        * We know the op was either Pfetch1 (type = 4) or Pstore1 (type = 10b)
        T ← ldf[Stack,15,1];     * we test Stack[13]: 0=> fetch, 1=> store
        Stkp ← RTEMP, lu ← T;    * point to count, test result to alu
        Stack ← (Stack) + 1, skip[alu # 0]; * count + 1; now test fetch/store
          T ← (PCXReg) - 1, goto[FPCOx];          * fetch; done with fixup
        Stack&-1, call[DecGlorp];         * source - 1
        Stack&+2, call[DecGlorp];         * dest - 1
        T ← (PCXReg) - 1, goto[FPCOx];

DecGlorp:       Stack ← (Stack) - 1, return;


FixPCOnly:      T ← (PCXReg) - 1,AT[FixDisp,0]; *normal PC fixup
FPCOx:  RTEMP ← T, skip[alu>=0];
          PCB ← (PCB) - (4c);    * PCX was 0, inst started in previous quadword
        lu ← (PCFReg) - (T); *test for PCX large, PCF small
        PCF ← RTEMP, skip[alu>=0]; *PCF is always PCX-1, only PCB is in doubt
          PCB ← (PCB) - (4c);


*Here PCB,PCF is correct pc to save for trap.  It will be done through KFCB.

StartMemTrap:   T ← SStkp, at[StartMemTrapLoc];
SMTrpx: RTEMP ← pPipeReg, task; *Point stkp to pipe registers
        Stkp ← RTEMP, RTEMP ← T;
        MemStat ← Normal;
        T ← (Stack&+1) and (177c);       *low 7 bits of VPage
        T ← (lsh[Stack&+1,7]) or (T);    *high 7 bits of VPage
        xfOTPReg ← (zero) or not (T);
        xfOTPReg ← (xfOTPReg) and not (140000c);
        stack&+3, task; *point to flags
        T ← ldf[Stack, 12, 1];  *Test Dirty':  0=> page fault.
        Stkp ← RTEMP, lu ← T;    *Restore stkp
        skip[ALU=0], LoadPage[7];
        T ← sWriteProtect, gotop[kfcr];
        T ← sPageFault, gotop[kfcr];

        END;
```

```
TITLE[GlobalDefs];
* Last Modified by Chang on October 10, 1979  7:29 PM, revised RDC & XW-codes
* Modified by Chang on September 17, 1979  5:36 PM, New RDC-codes
* Modified by Chang on September 7, 1979  4:27 PM, clean page 2
* Modified by Chang on August 23, 1979  2:42 PM, move Timer's regs & CSB
* Modified by Chang on August 13, 1979  3:03 PM, RDC Integration
* Modified by Chang on August 2, 1979  2:09 PM, Change XW-Tasks numbers
* Modified by Chang on July 6, 1979  6:07 PM, Two XWires
*Modified by Johnsson on June 15, 1979  3:18 PM, add Kernel registers (new Midas)
*Modified by Chang on May 27, 1979  4:41 PM, for Overlay booting (midas)
*Modified by Johnsson on May 14, 1979  8:53 AM
*Modified by Chang on May 10, 1979  8:27 AM
*Modified by Sandman on May 8, 1979  11:58 AM
*Modified by Johnsson on May 7, 1979  3:06 PM

IDF@[MidasBoot, , Set[MidasBoot,0]];
IDF@[XWire, , Set[XWire,0]];

*Task Numbers
Set[EOtask,12]; *Ethernet Output -- DO mode
Set[EItask,13]; *Ethernet Input -- DO mode
Set[EOtask2,6]; *Ethernet Output -- DO mode (second board only)
Set[EItask2,7]; *Ethernet Input -- DO mode (second board only)
Set[Dtask,10]; *IRDC
* MC[RdcTask, 11]; *SA4000, implies a set[RdcTask!, 11];
Set[RdcTask, 11];
Set[uiUTFPTASK,14]; *IUTFP
Set[TTask,16];
Set[RFTask,4]; *RS232 frame task
Set[RBTask,5]; *RS232 bit task

*********************** CSB Assignments
Set[XOStartCSB, 177640];        * Output CSB (board 1)
Set[XIStartCSB, 177660];        * Input CSB (board 1)
Set[XOStartCSB2, 177540];       * Output CSB (board 2)
Set[XIStartCSB2, 177560];       * Input CSB (board 2)
SET[RS232CSBLoc,177500];        * CSB location
Set[RDCCSBValue, 177620];       * RDC CSB
****************************************

*Page Assignments
Set[InitPage,16];
Set[InitPage1,2];
Set[InitPage2,1];
Set[LoadCSPage,1];
Set[TimerPage,0];
Set[TimerinitPage1,2];
Set[TimerinitPage2,2];

*NOTE - all initialization is crammed onto the same page.
Set[DiskInitPage,2]; *Throwaway init code for disk
        Set[DiskInitLoc,Add[Lshift[DiskInitPage,10],203]]; *Disk initialization
Set[DisplayInitPage,2]; *Throwaway init code for IUTFP
        Set[DisplayInitLoc,Add[Lshift[DisplayInitPage,10],207]]; *Display initialization
Set[RdcInitPage,2]; *Throwaway init code for SA4000
        Set[RdcInitBase,Add[Lshift[RdcInitPage,10],340]]; *SA4000
        Set[RdcInitLoc,Add[Lshift[RdcInitPage,10],340]]; *SA4000 Initialization
Set[EtherInitPage,2];
        Set[EtherInInitLoc,Add[Lshift[EtherInitPage,10],210]]; *Ethernet initialization
        Set[EtherOutInitLoc,Add[Lshift[EtherInitPage,10],212]]; *Ethernet initialization
* --> (2nd board only)
Set[EtherInitPage2,2];
        Set[EtherInInitLoc2,Add[Lshift[EtherInitPage,10],214]]; *2nd Ethernet initialization
        Set[EtherOutInitLoc2,Add[Lshift[EtherInitPage,10],216]]; *2nd Ethernet initialization
Set[EEPage,3]; *Ethernet microcode
Set[EOPage,3];
Set[EIPage,3];
Set[KeyPage,17]; *Keyboard translation table
        Set[KeyTable,add[lshift[keypage,10],140]];
        MC[KeyTableH, and@[keytable,7400]];
        MC[KeyTableL, and@[keytable,377]];

Set[opPage0,4]; *These cannot move easily, since the hardware forces the first instruction
Set[opPage1,5]; *of each bytecode to start at 2001 + (4 * opcode)
Set[opPage2,6];
Set[opPage3,7];
Set[MulDivPage,4];
Set[LRJPage,0];
Set[ClrDvPage,4];
Set[FaultPage1,14];

Set[DRPAGE,10]; *Disk microcde
Set[DRPAGE2,11]; *More disk microcode
set[bbp1,       11];    *bitblt page 1
Set[uiUTFPPage,12]; *Display microcode
Set[nePage,1];
Set[RdcPage, 13]; * SA4000 main task
%
************** page-assignment for RDC
Set[RdcPage1, 13]; *SA4000, main task
Set[RdcPage2, 13]; *SA4000, used to be page "RdcOverflow"
Set[RdcPage3, 17]; *SA4000, used to be page "RdcOverflow"
Set[RdcPage4, 17]; *SA4000, used to be page "RdcOverflow"
Set[RdcPage5, 17]; *SA4000, used to be page "RdcOverflow"
****************************
%
set[bbp2,       14];    *bitblt page 2
Set[xfPage1,15];
SET[prPage, 16];

Set[DBootDoneLoc,Add[lshift[InitPage1,10],375]];
```

```
Set[KFCRLoc,Add[lshift[oppage3,10],76]];
Set[P7TailLoc,Add[lshift[oppage3,10],27]];
Set[LoadGCLoc,Add[lshift[xfPage1,10],300]];
Set[neNoskipLoc,Add[lshift[nePage,10],273]];
Set[InitEndLoc,Add[lshift[TimerPage,10],20]];
Set[EEStartLoc,Add[lshift[EEPage,10],105]];
Set[StartMemTrapLoc,Add[lshift[FaultPage1,10],16]];
Set[PNIPBase,Add[lshift[TimerPage,10],156]];

* Pilot high resolution clock
SetTask[TTask];
RV[ClockLo, 50];
RV[ClockHi, 51];

* The following 5 definitions MUST MATCH THOSE IN RS232C MICROCODE!!!
* Also, code depends on RINotify value being zero!
MC[RINotify,0]; * Input bit notify least sig. byte address
MC[RONotify,1]; * Output bit notify least sig. byte address
MC[RPNotify,2]; * Poller notify least sig. byte address

RV[EONotify, 40];        * Register containing notify value for Ethernet
RV[EONotify2, 44];* Register containing notify value for 2nd Ethernet
RV[RFNotify,46];         * Register containing frame notify values
RV[RXNotify,47];         * Register containing bit notify values
* End of RS232 definitions

RV[RSImage,42];          *Image of RS232 hardware register

* Kernel registers
SetTask[17];
RV[RXALU,76];    *ALU result and SALUF
RV[RXAPC,75];    *APCTask&APC
RV[RXCTASK,74]; *CTASK.NCIA
RV[RXPPB,73];    *Page,Parity,BootReason
RV[RXSTK,72];    *Stackpointer
RV[RTMP,71];     *temporary
RV[FFault, 66]; *Flags tell what to do with fault
MC[FFaultAdd, 366]; *Address of FFault
SetTask[0];

*Register definitions for Nova and Mesa emulator
*The Mesa Stack.  These locations cannot move, since the hardware does overflow checking on
*these registers.
RV[Stack0, 1];
RV[Stack1, 2];
RV[Stack2, 3];
RV[Stack3, 4];
RV[Stack4, 5];
RV[Stack5, 6];
RV[Stack6, 7];
RV[Stack7, 10];

* Registers used to hold constants
RV[R400,15]; *constant 400
RV[AllOnes,16]; * -1
RV[RZero,17]; *0


*Nova central registers
RV[AC0,20]; *Quadword block for AC0-3
MC[pAC0,20]; *R address of AC0
RV[AC1,21];
RV[AC2,22];
RV[AC3,23];
RV[CARRY,24]; *Nova carry bit in bit 15d
RV[NWW,25];
MC[pNWW,25]; *pointer to NWW

*Base registers
RV[Nova,26]; *Base of Nova address space
RV[Novah,27];

RV[PCB,30]; *PC base register pair
RV[PCBh,31];

RV[PC, 30];
RV[PChi, 31];

RV[DMA,32]; *temporary base register used by CONVERT
RV[DMAh,33];

RV[SMA,34]; *temporary base register used by CONVERT
RV[SMAh,35];

RV[MDS, 36];
RV[MDShi, 37];

RV[GLOBAL, 54];          * must be quad aligned
RV[GLOBALhi, 55];
RV[xfMY, 56];            * must be GLOBAL + 2
RV[xfMX, 57];

RV[LPdest, 56];          * Long BLT
RV[LPdesthi, 57];

RV[CODE, 60];
RV[CODEhi, 61];

RV[LOCAL, 64];
RV[LOCALhi, 65];
```

```
RV[LP,66]; *long pointer base pair
RV[LPhi,67];

RV[PBase, 26];  * PSB base register
RV[PBasehi, 27];

RV[Queue1,44]; * Queue base register for process machinery
RV[Queue1hi, 45];
RV[Queue2, 46]; * Queue base register for process machinery;
RV[Queue2hi, 47];

*Buffers
RV[IBUF,40]; *4 word instruction buffer (RM 40-43)
RV[IBuf1, 41];
RV[IBuf2, 42];
RV[IBuf3, 43];

RV[xBuf,44]; *Quadword temporary buffer (RM 44 - 47)
	MC[pxBuf,44]; *pointer to xBuf
RV[xBuf1, 45];
RV[xBuf2, 46];
RV[xBuf3, 47];

*Time registers for process timoout;
RV[CurrentTime, 62];
RV[TickCount, 63];

*Other
RV[xfTsi, 32];
RV[xfRlink, 33];
RV[RLink, 33];
RV[xfGfiWord, 34]; *holds word 0 of global frame
RV[xfRsav, 35];
RV[EfAd,50]; *Nova Effective address
RV[MemStat, 50];
RV[RCNT,51]; *used by MUL and DIV
RV[xfFrame, 51];
RV[RTEMP,52];
RV[RTEMP1,53]; *used by Interrupt Test
RV[Result,53]; *temporary
RV[intRTN,54]; *used by Interrupt test
RV[INTX,55]; *used by Interrupt test
RV[xnXH,56]; *temporary used by CONVERT
RV[WW,56];
RV[ACTIVE,57]; *must be WW+1
RV[xnDest,57]; *temporary used by CONVERT
RV[XBI,57]; *register used as temporary when initializing DB as an index into xBuf
RV[xfBrkByte, 74];

*Process registers
RV[Process, 73];
RV[MQ, 66];
RV[PRFlags, 71];
RV[Prev, 70];
RV[QTemp, 12];
RV[QTemphi, 13];
RV[PRTime, 72];
RV[RTemp2, 67];
RV[ITemp, 12];
RV[ITemp1, 13];
RV[IntType, 14];
RV[IntLevel, 57];
RV[EMLink, 10];

RV[xfTemp, 72];
RV[xfTemp1, 73];
RV[xfTemp2, 66];
RV[xfCount, 67];
RV[xfATPreg, 71];
RV[xfXTPreg, 70];
RV[xfOTPreg, 11];
RV[EHostReg,75]; *Ethernet host register (accessed by Ethernet code during SIO)
	MC[pEHostRegx, 74]; *pointer used during Ethernet initialization (points to 74 because
*INPUT to the stack does a push
RV[xfWDC, 76];
RV[xfXTSreg, 77];


*bitblt registers
rv[bbDEST,		44];*Quad-word buffer coincident with xBuf
rv[bbRTEMslx,		44];
rv[bbRTEMsty,		45];*paired with bbRTEMslx
rv[bbRTEMdlx,		46];
rv[bbRTEMdty,		47];*paired with bbRTEMdlx

rv[bbSOURCE,		70];*Quad-word buffer

rv[bbSrcQAddrLo,	32];*map base reg
rv[bbSrcQAddrHi,	33];*paired with bbSrcQAddrLo

rv[bbDestQAddrLo,	56];*map base reg
rv[bbDestQAddrHi,	57];*paired with bbDestQAddrLo

rv[bbSBCA,		4];
rv[bbGRY,		4];
rv[bbSBMR,		5];*paired with bbSBCA

rv[bbDBCA,		6];
rv[bbGrayCnt,		6];
rv[bbDBMR,		7];*paired with bbDBCA
```

```
rv[bbMinusItemWidth,        66];
rv[bbItemWidth,             66];
rv[bbItemsRemainingMinus2,67];*paired with bbItemWidth
rv[bbItemsRemainingMinus1,67];
rv[bbItemsRemaining,        67];

*registers which need not be paired
rv[bbSrcStartBitLo,         12];
rv[bbSrcStartBitHi,         13];
rv[bbDestStartBitLo,        14];
rv[bbDestStartBitHi,        20]; *coincident with AC0

rv[bbMinusSDNonOverlap,     24];
rv[bbSDNonOverlap,          24]; *coincident with CARRY
rv[bbMinusBitsRemaining,    51];
rv[bbMinusNumBitsTran,      53];
rv[bbFunction,              11];

RV[xCNT,52];
RV[DevIndex,62];


*Constants

MC[IntPendingBit, 10];

MC[xfAV, 1000];
MC[xfSDOffset, 100];
MC[xfGFT, 1400];

MC[Normal, 0];  * Things in MemStat
MC[FreeFrame, 10];
MC[EarlyXfer, 4];
MC[BltFixup, 3];
MC[BltLFixup, 2];
MC[XferFixup, 1];

*Frame formats
MC[xfPcOffset, 1];        * in L
MC[xfRetLinkOffset, 2]; * in L
MC[LocalZeroOffset, 4]; * in L
MC[xfGFiOffset, 0];       * in G
MC[GlobalZeroOffset, 3];          * in G

*StateVector format
MC[StkPOffset, 10];
MC[DestOffset, 11];
MC[SourceOffset, 12];

*SD indicies
MC[sStackError, 2];
MC[sWakeupError, 3];
MC[sXferTrap, 4];
MC[sUnimplemented,5];
MC[sAllocListEmpty, 6];
MC[sControlFault, 7];
MC[sCsegSwappedOut, 10];
MC[sPageFault, 11];
MC[sWriteProtect, 12];
MC[sUnbound, 13];
MC[sZeroDivisor, 14];
MC[sDivideCheck, 15];
MC[sHardwareError, 16];
MC[sProcessTrap, 17];
MC[sBoundsFault, 20];
MC[sPointerFault, 21];

MC[FirstProcess, 75];
MC[LastProcess, 76];
MC[FirstStateVector, 77];

MC[CurrentPSB, 21];
MC[ReadyQ, 22];
MC[CurrentState, 23];
MC[ReadyQhi, 1];

MC[IntStopPC,25];         *Nova entry point constants
MC[StopStopPC,26];
MC[MEStopPC,27];
MC[MXDStopPC,30];
MC[MREStopPC,31];
MC[MXWStopPC,32];
MC[NOTIFYStopPC,33];
MC[BCASTStopPC,34];
MC[REQUEUEStopPC,35];

* RS232 SIO address constants
Set[RS232SIOLoc, add[LShift[EEPage,10], 370]];

%
********** move the followings to D0Lang.mc
*If mode not defined, make it Pilot
IDF@[AltoMode,,SET[AltoMode, 0]];

*This statement defines comments *// is Alto and *= is Pilot
IFE@[AltoMode,0,COMCHAR@[#],COMCHAR@[=]];

*Print Message telling Mode
IFE@[AltoMode,0,ER@[Pilot.3.0.Microcode],ER@[Alto/Mesa.5.0.D0.Microcode]];

*Macros
```

```
M@[OPCODE, AT[2001,LSHIFT[#1,2]]];
*Cycler-masker functions
M@[FIXVA, BS@[3] FZ@[341] #1];
M@[Form1,LDF[#1,17,1]];
M@[Form2, BS@[3] FZ@[342] #1];
M@[Form3,LDF[#1,16,2]];
M@[Form4, BS@[3] FZ@[343] #1];
M@[FormMinus4, BS@[3] FZ@[351] #1];
%

        End;
```

```
BUILTIN[insert,24];
insert[d0lang];
NOMIDASINIT;
LANGVERSION;
MULTDIB;

insert[GlobalDefs]; *task and page assignments

TITLE[Initialization];

*last edit by Chang, October 11, 1979  10:03 AM, revised RDC & XW-codes
* edit by Johnsson, October 9, 1979  4:06 PM, AR 1829 - PNIP
* edit by Kennedy, October 3, 1979  9:22 AM, Update MP Codes
* edit by Chang, September 17, 1979  7:00 PM New RDC-codes
* edit by Chang, September 12, 1979  2:13 PM RDC in ALTO-mode
* edit by Chang, August 22, 1979  1:29 PM move Timer's regs
* edit by Chang, August 17, 1979  8:42 AM RDC Integration
* edit by Chang, August 2, 1979  3:53 PM Add 2nd EtherBoard
* edit by Johnsson, June 13, 1979  4:53 PM new FFault register
*edit by Chang, June 3, 1979  4:11 PM Overlay booting
*edit by Johnsson, May 15, 1979  3:27 PM, PNIP fix
*edit by Chang, May 10, 1979  8:31 AM, add new Ethernet ID
*edit by Sandman, April 6, 1979  3:50 PM

*Modified March 6, 1979 by CPT. Added fault handling

IMRESERVE[0,0,100];
* IFE@[InitPage,1, IMRESERVE[0,100,17], IMRESERVE[1,0,100]];
IMRESERVE[1,0,100];
IMRESERVE[2,0,100];
IMRESERVE[17,377,1];


*Registers for IMAP
RV[xmad0,22]; *base register
RV[xmad1,23];
RV[xmbuf0,24]; *quadword buffer for XMap and PFetch4
RV[rbuf0,24];
RV[xmbuf1,25];
RV[rbuf1,25];
rv[xmbuf2,26];
RV[rbuf2,26];
RV[xmbuf3,27];
RV[rbuf3,27];
RV[wbuf0,30]; *quadword buffer for PStore4
RV[wbuf1,31];
RV[wbuf2,32];
RV[wbuf3,33];
RV[rlink0,34]; *subroutine return link
RV[MapEntry,35]; *current map location
RV[ZPage,35]; *page being cleared
RV[RealPage,36]; *current real storage page
RV[ZWord,36];
RV[PageCount,37]; *count of available real pages in system
RV[CompFlag,40];
RV[BootType,0]; * even => hard boot, odd => soft boot

*Registers for other sections of initialization
RV[xCNT,20]; *used everywhere
RV[DevIndex,21]; *used in DeviceInit
MC[pDX,21]; *pointer to DevIndex
RV[contemp,22]; *used in DeviceInit
RV[assigned,23]; *used in DeviceInit
RV[initpc,24]; *used in DeviceInit
RV[initr0,40]; *used in DiskBoot
RV[initr1,41]; *used in DiskBoot
RV[initr2,42]; *used in DiskBoot
RV[initr3,43]; *used in DiskBoot
RV[ErrorCnt,44]; *used in DiskBoot
RV[ErrorCountx,45]; *used in DiskBoot

* MC[tmr38conreg,324]; *RM 324 holds 38usec timer restart constant
MC[tmr38conreg,354]; *RM 354 holds 38usec timer restart constant


*Maintenance Panel Normal Operation Codes:
MC[StartMapInit,1274]; *700d
MC[StartDeviceInit,1306]; *710d
MC[StartDiskBoot,1320]; *720d
MC[SystemRunning,1476]; *830d

*Maintenance Panel Failure Codes:
MC[NotEnoughMemory,1275]; *701d
MC[BadMap,1276]; *702d
MC[NoDiskStatus,1321]; *721d
MC[BadBoot,1322]; *722d

SETTASK[0];

MC[NextDiskAddr,237];
RV[BootDiskAddr,37];
SET[InitBase,lshift[InitPage,10]];
  SET[HardStart, ADD[InitBase,1]];
  SET[SoftStart, ADD[InitBase,2]];
MC[InitLoc,InitBase];

SET[Qloc,ADD[InitBase,5]];
MC[QxL,AND@[Qloc,377]];
MC[QxH,OR@[150000,AND@[Qloc,7400]]];

SET[QretLoc,Add[InitBase,7]];
```

```
MC[QRetL,AND@[QretLoc,377]];
MC[QretH,AND@[QretLoc,7400]];

        ONPAGE[InitPage];
*Machine initialization begins here
GO:
START:  xCNT←InitLoc, at[HardStart]; *send control to "Qtask" in task 0
        apc&apctask←xCNT, goto[initRET];

SoftBoot:
        BootType ← 1c, goto[RegInit1], at[SoftStart];    * This is a soft boot;

Qtask:  xCNT ← QxL, AT[InitBase]; *Quiesce tasks 15b - 1
        xCNT ← (xCNT) or (QxH);
        DevIndex ← pDX;
        stkp ← DevIndex;
        DevIndex ← QRetL;
        DevIndex ← (DevIndex) or (QRetH);
Qloop:  APC&APCTASK ← xCNT;
initRET:          return; *goes to Qx

Qx:
  APC&APCTASK ← stack,call[initRET], AT[Qloc]; *Notify comes here. Leave task's TPC pointing at Qxy.
Qxy:    goto[initRET]; *must spend two instructions in the task


Qret:   lu ← ldf[xCNT,0,3], AT[QretLoc]; *xCNT points to this location
        xCNT ← (xCNT) - (10000C), dblgoto[ZapDevices,Qloop,ALU=0];

ZapDevices:
        BootType ← 0c;  * This is a hard boot
        T ← 177400C;
        xCNT ← 0c;
ZapDloop:         OUTPUT[xCNT];    *send a 0 to all registers of all devices, hopefully quiescing them
        T ← (zero) + (T) + 1;
        dblgoto[ZapDloop,DoMap,ALU<0];

*The following section tests the map as a memory, then determines the
*amount of real storage in the system and sets up the first
*N map entries to point to this storage (remaining map entries
*are initialized to VACANT), then clears storage.
DoMap:  loadpage[0];
        T ← StartMapInit, CallP[PNIP];

IMAP:   CompFlag ← (zero)-1,goto[imCompx];
imAloop:
        T ← (CompFlag) xor (T);
        xmbuf0 ← T, call[imWriteMap];
        T ← MapEntry;
        T ← (CompFlag) xor (T), call[imReadMap];
        MapEntry ← T ← (MapEntry) + 1;
        goto[imAloop,nocarry];
        MapEntry ← 140000C;
imRloop:
        T ← MapEntry;
        T ← (CompFlag) xor (T), call[imReadMap];
        MapEntry ← T ← (MapEntry) + 1;
        lu ← CompFlag, goto[imRloop,nocarry];
        goto[.+2,ALU=0], CompFlag ← (zero);
imCompx:
        goto[imAloop], MapEntry ← T ← 140000C;


        RealPage ← (10000C); *max real page +1
        MapEntry ← 140000C; *carries beyond max VM cause ALUCY
        PageCount ← 0C;
        rlink0 ← FFaultAdd; *location in fault handler
        stkp ← rlink0;

*fill first quadword of each real page with its page number and some constants.
*go through real memory backwards so that hole in 96k modules will not
*screw up non-hole banks.

        wbuf1 ← 326C; *random constant
        wbuf2 ← 134000C;
        wbuf3 ← (zero) ;
imFloop:
        RealPage ← T ←  (RealPage)-1;
        xmbuf0 ← T, goto[.+2,ALU>=0];
        T ← RealPage ← 170000C, goto[imTloop];
        wbuf0 ← T;
        call[imWriteMap];
        PStore4[xmad0,wbuf0,0], goto[imFloop];

*during this phase, we sweep upward through real storage and the map, and
*use any real pages discovered.

imTloop:
        xmbuf0 ← T;
        xmbuf0 ← (xmbuf0) and not (170000C), call[imWriteMap]; *set base reg to point to MapEntry,
*set MapEntry to point to RealPage.  Set all map flags off.
        nop;
        call[.+2], stack ← (Stack) or (100000c);          *turn on fault handler
imFault:
        goto[imPageBad], stack ← (Stack) and not (100000c); *get here on a fault - turn off fault handler
        PFetch4[xmad0,rbuf0,0]; *fetch.  Will cause fault if page is bad
        T ← rbuf0;
        lu ← (ldf[RealPage,4,14]) xor (T);
        goto[.+2, ALU = 0], stack ← (Stack) and not (100000c);   *turn off fault handler
        goto[imPageBad]; *page number didn't compare
        T ← (rbuf1) xor (326C); *a final check against constants
```

```
        rbuf2 ← (rbuf2) xor (134000C);
        T ← (rbuf2) or (T);
        T ← (rbuf3) or (T);
        dblgoto[imPageGood, imPageBad, ALU=0];

imPageGood:
        MapEntry ← (MapEntry) + 1;
        PageCount ← (PageCount) + 1;
imPageBad:  T ← RealPage ← (RealPage) + 1;
        goto[imTloop, nocarry]; *done with all of real memory?
        wbuf1 ← 0C; *clear wbuf1 in preparation for core zap.
imMarkVacant:
        xmbuf0 ← 60000C; *page vacant
        Call[imWriteMap];
        MapEntry ← (MapEntry) + 1;
        T ← PageCount, goto[imMarkVacant, nocarry]; *done with all map entries?

imCoreZap:
        wbuf3 ← 0C;
        wbuf2 ← 0c;
        Zpage ← T;

imZapLoop:
        ZPage ← (ZPage) -1;
        T ← lhmask[ZPage], goto[imDone, ALU<0];
        xmad1 ← T; *set up a base register for the page
        T ← lsh[Zpage,10];
        xmad0 ← T;
        ZWord ← 400C, call[imZPloop];

imZPloop:
        Zword ← T ← (Zword) - (4C);
        goto[imZapLoop, ALU<0];
        PStore4[xmad0, wbuf0], return;

imDone:
        lu ← (PageCount) -(400c); *don't try to run with less than 64K
        T ← NotEnoughMemory, goto[InitFail, ALU<0];
        goto[RegInit1];

InitFail:
        loadpage[0];
        callp[PNIP];
        goto[START];

*SUBROUTINE imWriteMap writes the data in xmbuf0
*into map location MapEntry
imWriteMap:
        T ← (MapEntry) and not (140000C);
        xmad1 ← T;
        xmad1 ← (xmad1) and not (377C);
        T ← lsh[MapEntry,10];
        xmad0 ← T;
        Xmap[xmad0,xmbuf0,0];
        xmbuf0 ← xmbuf0, return; *interlock

*SUBROUTINE imReadMap reads one entry from MapEntry
*into rbuf0, then compares it with (MapEntry xor CompFlag)
imReadMap:
        xmbuf0 ← T, usectask;
        T ← apc&apctask;
        rlink0 ← T, call[imWriteMap];
        T ← lsh[xmbuf3, 10]; *flags, card, blk.0 bits
        rbuf0 ← T;
        T ← (xmbuf1) and (377C);
        rbuf0 ← (rbuf0) xor (T);
        T ← MapEntry;
        T ← (CompFlag) xor (T);
        lu ← (rbuf0) xnor (T); *note, Map data is complemented, so we xnor
        goto[imGoodEntry, ALU=0];
imBadMap:
        T ← BadMap, goto[InitFail]; *Some map entry was bad

imGoodEntry:
        apc&apctask ← rlink0, goto[initRET];
```

```
*We have initialized the map and memory.  Before initializing and
*starting devices, set up the R memory locations that
*must be valid.

RegInit1:
        Nova ← zero;
        Novah ← zero;
        DMAh ← zero;
        R400 ← (400c);
        AllOnes ← (zero)-1;
        RZero ← zero;
        xCNT ← NextDiskAddr;
        stkp←xCNT;
        t←stack;                        *get next disk address from reg 237
        BootDiskAddr ← t; * save next disk address in reg 37

* RS232 initialization..fake RS232 stop (task 16, location RS232SIO+1)
        xCNT ← OR@[LSHIFT[TTask,14],AND@[007400,ADD[RS232SIOLoc,1]]]C;
        xCNT ← (xCNT) OR (AND@[377,ADD[RS232SIOLoc,1]]C);
        APC&APCTask ← xCNT, TASK;
        Return;

*Initialize the 38usec timer
        xCNT ← (tmr38conreg);
        stkp ← xCNT, task ;
        stack ← (50000c) ;
        stack ← (stack) or (156c) ; *simple timer,value 6,slot 16
                                    * with 100ns clock, period is
                                    * 4*16*100*6ns = 38.4 us
                                    * Alto is nominally 38.08 us
        LoadPage[InitPage2];
        loadtimer[stack], goto[DeviceInit] ;
```

```
*Find and initialize all I/O devices:
*The idea is to use RM 40-57 as a "slot table" with one entry
*per potential I/O controller.  First the table is filled
*with dummy controller addresses, and these are clocked out
*to the controllers.  Then, each controller is interrogated
*for it's Device ID, and these names are put in the table.
*Then, for each slot, the device ID is looked up in a table
*(in the control store) of potential devices, and if a match
*is found, the entry from the device table is put into the
*slot table.  A device table entry consists of the uPC value
*of the device's initialization routine (12 bits), and the
*task number for the controller.
*When all slots have been looked up, the task numbers are
*clocked out to the controllers, and the associated initialization
*routines are called in turn.

*To add a new controller to the system, it is only necessary
*to add an entry to the device table (and add the controller's
*microcode).

*first, a macro to allow nice formatting of the device table entries...
m@[dtab,DATA[(LH[#1],RH[LSHIFT[#2,4],#3],RSEL2@[dp[#1,#2,#3]],at[dtabloc])]SET[dtabloc,ADD[dtabloc,1]]];

*also, we must make the parity of an entry correct, or Midas will correct it for us...
m@[dp,set[dpx,xor@[1,#1,#2,#3]]set[dpx,xor@[dpx,rshift[dpx,10]]]set[dpx,xor@[dpx,rshift[dpx,4]]]
  set[dpx,xor@[dpx,rshift[dpx,2]]]set[dpx,xor@[dpx,rshift[dpx,1]]]
   set[dpx,and@[1,dpx]]dpx];

SET[DtabBase,add[InitBase,100]];
SET[dtabloc,DtabBase];

*Here is the device table
dtab[2003,DiskInitLoc,DTask]; *IRDC ID = 2003b,
dtab[2402,EtherInInitLoc,EITask]; *Ethernet input = 2402b
dtab[2401,EtherOutInitLoc,EOTask]; *Old Ethernet output = 2401b
dtab[3400,EtherInInitLoc,EITask]; *Ethernet input = 3400b
dtab[3000,EtherOutInitLoc,EOTask]; *Ethernet output=3000b
dtab[6400,EtherInInitLoc,EITask]; * XWire input
dtab[6000,EtherOutInitLoc,EOTask]; * XWire output
*=******* Start of Pilot Code *******************************************
dtab[2402,EtherInInitLoc2,EITask2]; *2nd Old Ethernet input            *
dtab[2401,EtherOutInitLoc2,EOTask2]; *2nd Old Ethernet Output        *
dtab[3400,EtherInInitLoc2,EITask2]; *2nd Ethernet input              *
dtab[3000,EtherOutInitLoc2,EOTask2]; *2nd Ethernet Output            *
dtab[6400,EtherInInitLoc2,EITask2]; *2nd XWire input                 *
dtab[6000,EtherOutInitLoc2,EOTask2]; *2nd XWire Output      *
*=******* End of Pilot Code *******************************************
dtab[1417, RdcInitLoc, RdcTask]; *SA4000
dtab[1407, RdcInitLoc, RdcTask]; *SA4000
dtab[2417, RdcInitLoc, RdcTask]; *SA4000
dtab[2407, RdcInitLoc, RdcTask]; *SA4000
dtab[411,DisplayInitLoc,uiUTFPTASK]; *IUTFP ID = 411b
dtab[0,0,0];     *final entry in the table must be zero

        ONPAGE[InitPage2];
DeviceInit: loadpage[0];
        T← StartDeviceInit,callp[PNIP];
        xCNT ← 57c;
        stkp ← xCNT, xCNT ← T ← (xCNT)+1, call[DI1];

*stkp = 57b, xCNT = T = 60b here.

*Write 60-77 into RM 40-57
DI1:    stack&+1 ← T;
        lu ← (xCNT) xor (77C);
        xCNT ← T ← (xCNT) + 1, goto[DI2, alu=0];
        return; *return to DI1

*Send dummy controller addresses to devices
DI2:    xCNT ← 57c;
        stkp ← xCNT, call[ClockOutPattern];

*stkp = 57b here
        xCNT ← T ← 177400c, call[DI3]; *Read controller ID's from register 0 of all devices
DI3:    INPUT[stack];
        nop; *allow xCNT & T to be written
        xCNT ← T ← (xCNT) + (20C), goto[DI4,R>=0]; *advance to next device
devRET: return; *return to DI3

*Look up each slot table entry in the device table
DI4:    xCNT ← 40c;
        stkp ← xCNT;
        xCNT ← 17c;
        assigned ← 0c; * Bit mask of asigned tasks
DI4x:   DevIndex ← 0c; *base of device table in control store
DI4y:   T ← DevIndex, call[GetCon]; *get a device ID from the device table
        lu ← (stack) xor (T); *compare with the slot table entry
        goto[DevFound, alu=0], lu ← T;
DI4u:   goto[DI4y, ALU#0], DevIndex ← (DevIndex) + (2c); *check for end of table (zero entry)
        stack ← 17c; *end of table reached without match - set slot's task to 17 (unused)
DI4z:   xCNT ← (xCNT)-1;        *check for all slots processed
        stack&-1, dblgoto[DI4x, DI5, ALU>=0]; *set to next slot

DevFound:       nop; *allocation constraint
        T ← (DevIndex)+1, call[GetCon];
        initpc ← T;
        T ← LDF[initpc,14,4]; * Task number about to get assigned
        contemp ← T;
        contemp ← LSH[contemp,4];
```

```
        CycleControl ← contemp;
        T ← WFA[AllOnes];
        LU ← (assigned) AND (T);
        SKIP[ALU=0];
        GOTO[DI4u], LU ← 1C;  * Keep looking, task already assigned
        assigned ← (assigned) OR (T);
        T ← initpc;
*//******* Start of Alto Code ********************************************
        stack ← T, goto[DI4z]; *replace slot table entry with device table entry
*//******* End of Alto Code ********************************************

*=******* Start of Pilot Code ********************************************
        stack ← T;*replace slot table entry with device table entry
        rbuf1 ← T; * device table entry contains task assignment
        T ← StkP;
        rbuf0 ← T; * save the device pointer
        rbuf1 ← (rbuf1) AND (17C); * get the task assignment
        T ← (LSH[rbuf1, 4]);
        rbuf1 ← T;
        StkP ← rbuf1;
        T ← (rbuf1) + (177400C);
        rbuf0 ← (rbuf0) XOR (377C);
        Stack ← T;
        StkP ← rbuf0, Goto[DI4z];
*=******* End of Pilot Code ********************************************

*Clock out new controller ID's
DI5:    xCNT ← 57c;
        stkp ← xCNT, call[ClockOutPattern];

*Call all the Init routines
        xCNT ← 17c, call[DI6]; *do call to set up TPC for loop below
DI6:    xCNT ← (xCNT)-1, goto[DI7, R<0];
        lu ← ldf[(stack),4,14]; *check for Init PC = 0 (no initialization required)
        goto[.+3, ALU=0], T ← Stack&-1;
        DevIndex ← T; *set up to call Init routine for device
        APC&APCTask ← DevIndex; *call Init routine for controller - returns to DI6
        return;

DI7:    LoadPage[InitPage];
        ErrorCnt ← (10c), goto[BootEmulators]; *Set up retry count for disk boot

ClockOutPattern:
        usectask;
        T ← APC&APCTASK;
        rlink0 ← T;
        xCNT ← 17c; *go through the slot table backwards
COP1:   DevIndex ← 2c; *DevIndex used for loop count
COP2:   T ← (stack) xor (1C); *complement bit
        GENSRCLOCK;     *send bit
        stack ← rcy[stack,1]; *get next bit
        DevIndex ← (DevIndex) - 1, goto[COP2, r>=0];
        xCNT ← (xCNT) -1; *all slot table entries done?
        stack&-1, goto[COP1, alu>=0]; *get next word
        APC&APCTASK ← rlink0, goto[devRET];

GetCon: contemp ← T; *word index into Dtab
        contemp ← (contemp) + (AND@[lshift[DtabBase,1], 17400]C);
        contemp ← (contemp) or (AND@[lshift[DtabBase,1], 377]C);
        contemp ← rsh[contemp,1]; *instruction address in CS
        T ← (ldf[AllOnes,17,1]) and (T); *low bit tells which half
        APC&APCTask ← contemp;
        READCS;
        T ← CSDATA, return, AT[lshift[InitPage2,10],100]; *location must be even

        OnPage[InitPage];
BootEmulators:
        lu ← BootType, goto[BootSecondBlock, R even];
*       loadpage[InitPage1];
*       gotop[DiskBootDone];
        t ← xfTemp, loadpage[0];
        gotop[LRJloop];

*Read disk Sector 0 into page 0 (starting at location 1).
DiskBoot:
        loadpage[0];
        T← StartDiskBoot, callp[PNIP];
        initr0 ← 0c; *word 520 - not used by disk
        initr1 ← 0c; *word 521 - IOCB pointer
        initr2 ← 0c; *word 522 - disk status
        initr3 ← 0c; *word 523 = -1 to force a seek
        t←(R400)or(120c);
        pstore4[Nova,initr0],CALL[initRET]; *clear KBLK at 521-523 (520 is also cleared)

*Set up the IOCB at 1000b
        DMA←1000C; *set up base register for DCB
        PSTORE2[DMA,initr0,0],CALL[initRET]; *clear pointer to next DCB at 1000b, status at 1001b

        xCNT ← 44000C; *disk command goes at location 1002 (read, read, read)
        PSTORE1[DMA,xCNT,2], CALL[initRET];

        xCNT ← 2000C; *header goes at 2000 (unlike ALTO)
        PSTORE1[DMA,xCNT,3], CALL[initRET];

        xCNT ← 400C; *label goes at 402 (like ALTO)
        xCNT ← (xCNT) + (2C);
        PSTORE1[DMA,xCNT,4], CALL[initRET];

        xCNT ← 1C;
        PSTORE1[DMA,xCNT,5], CALL[initRET]; *data goes at 1
```

```
*Since we will initialize the interrupt system later, we do not need to worry about 1006 or 1007
*Location 1008 is unused

        xCNT ← 1C;
        PSTORE1[DMA,xCNT,11], CALL[initRET]; *Disk address 0, with RESTORE bit at 1009
        T ← 2000C;
        T ← (zero) + (T) +1;
        PSTORE1[Nova,xCNT], call[initRET]; *smash header at 2001b


*Start the disk
        initr1 ← 1000c; *word 521 - IOCB pointer
        initr3 ← (zero)-1; *word 523 = -1 to force a seek
        T ← (R400) + (120C); *store block above in 520-523 (words 0 and 2 are zero)
        PSTORE4[Nova,initr0], call[initRET];

*Wait for the disk to store good status in the DCB, retry if status is bad
        ErrorCountX ← 20c;
DWSet:  DevIndex ← (zero)-1; *loop count for status wait
DiskWait:       Pfetch1[DMA,xCNT,1]; *fetch status word at 1001b
        T ← 17C, call[initRET];
        lu ← (ldf[xCNT,4,4]) xor (T);
        goto[StatusStored,ALU=0], lu ← ldf[xCNT,10,10];
        DevIndex ← (DevIndex)-1;
        goto[DiskWait,ALU#0];
        ErrorCountx ← (ErrorCountx) -1;
        goto[DWSet,ALU>=0];

NoStatus:
        T ← NoDiskStatus, goto[InitFail]; *timed out waiting for disk to store status

StatusStored:   dblgoto[GoodStatus,IncErCnt,ALU=0],FREEZERESULT;

GoodStatus:  T← (2000C) ; *check header at 2001
        T ← (zero) + (T) + 1;
        PFETCH1[Nova,xCNT],CALL[initRET];
        LU←xCNT;
IncErCnt:
        skip[ALU#0], ErrorCnt ← (ErrorCnt)-1;
          goto[LoadOtherCS];
        goto[BootFail,ALU < 0]; *bad header, try again
        goto[DiskBoot];
BootFail:
        T ← BadBoot, goto[InitFail]; *read 10 times, but header was wrong


*SUBROUTINE PNIP puts the number in T into the maintenance panel
*It will be used after initialization is complete
*Does not task unless called from task 0
        ONPAGE[0];

PNIP:   usectask, RTEMP ← T, at[PNIPBase,20];
        T ← APC&APCTask, at[PNIPBase,17];
        RCNT ← T, ClearMPanel, call[.+1], at[PNIPBase,0];
PNloop: RTEMP1 ← 4C, at[PNIPBase,1];
        RTEMP1 ← (RTEMP1)-1, dblgoto[.+1,.,ALU<0], at[PNIPBase,6];
        RTEMP  ← (RTEMP) - 1, at[PNIPBase,7];
        lu ← ldf[RCNT,0,4], goto[PNdone, ALU<0], at[PNIPBase,16];
        skip[alu#0], at[PNIPBase,4];
          IncMPanel, return, at[PNIPBase,2]; * task 0, tasking ok
        IncMPanel, goto[PNloop], at[PNIPBase,3]; * task #0, tasking not allowed
PNdone: APC&APCTask ← RCNT, at[PNIPBase,5];
        return, at[PNIPBase,15];

*SUBROUTINE DoInt ORs the bits from T into NWW and sets
*IntPending.  Uses registers 0 and 1 in whatever task calls.
*This code must not allow task switches.
*If T[0] = 1 on entry, the routine does a tasking return, else it returns without tasking
        ONPAGE[0];
RV[IntTemp1,0];
RV[IntTemp2,1];

DoInt:  IntTemp1 ← T;
        IntTemp2 ← pNWW, skip[ALU<0];
        T ← 377c, goto[.+2];
        T ← (Zero) - 1;
        T ← (Stkp) xor (T);
        Stkp ← IntTemp2, IntTemp2 ← T;
        T ← (IntTemp1) and not (100000C);
        IntTemp1 ← 342c;                * RSImage in Kernel
        Stack ← (Stack) or (T);         * set bits in NWW
        Stkp ← IntTemp1, skip[ALU=0];
        T ← Stack ← (Stack) or (IntPendingBit), goto[.+2];
        T ← Stack;
        Stkp ← IntTemp2, skip[R<0];
        usectask;
        RS232 ← T, return;              * set IntPending



%
*Very simple fault handler, used only during initialization.

*If IMAP or DEVICEINIT expects an error, it will set up its
*TPC to point to the instruction to be executed if an error
*occurs, and will set RM 354 # 0.  When the hardware gets a fault,
*the Kernel will send control to 120 if RXPPB[4:7] # 0.  This code
*checks RXPPB to determine the type of error.  If it is a memory
*error and RM 354 # 0, it does RESETMEMERRS and RETURNs.  If not, it notifies
*location 7514, task 17 (Kernel go3 overlay location BPSEND).
```

```
SETTASK[17];

RV[RXPPB,23]; *Page, parity, bootreason register in kernel
RV[RData, 43]; *Register holding data to be sent to Midas
RV[FFAULT,54]; *zero means send all faults to Midas, nonzero means return on memory errors


        OnPage[0];

FAULTxx:        lu ← LDF[RXPPB,7,1], AT[120];
        goto[MemErr, alu#0],lu ← FFAULT;
*Error was not a memory error
MidasFault:  RData ← 177400c; *Notify kernel at 7514b
        RData ← (RData) OR (114C);
        APC&APCTASK ← RData;
        RETURN, RData ← 40400C; *Message for Midas = 101b

MemErr: goto[.+2,ALU#0],RESETMEMERRS;
        goto[MidasFault];
        RETURN; *back to the task that caused the fault
%

        END;
```

```
INSERT[DOLANG];
NOMIDASINIT;
TITLE[kernel];

%
Modified June 28, 1979 by RJ. (FFault ← 40000c)
Modified June 28, 1979 by CT.

This kernel consists of two parts:
1) A section that occupies part of pages 0 and 17, runs at task 17, and handles
all communication with Midas with the exception of Mouse halt testing
(which is done by a timer).  This section refreshes the memory frequently
without using a timer.

2) A section that occupies part of page 16, runs at task 16, and handles
kernel initialization, normal memory refresh and Mouse halt testing while a program is running.

The idea is that if you have a simple program, you can use both parts of
this kernel, and you will get minimal memory refresh and mouse halt testing.
If you need something more complex, you overwrite the stuff on page 16,
but in this case you must supply the code for mouse halt testing.
%
SETTASK[17];

RV[REFR,77];     *memory refresh address

*The following registers hold the volatile state of the processor on a fault:
RV[RXALU,76];    *ALU result and SALUF
RV[RXAPC,75];    *APCTask&APC
RV[RXCTASK,74];  *CTASK.NCIA
RV[RXPPB,73];    *Page,Parity,BootReason
RV[RXSTK,72];    *Stackpointer

RV[RTMP,71];     *temporary

*The following registers are used for D0-Midas communication (RTMP is also used):
RV[RWSTAT,70];   *status register
RV[RDATA,67];    *holds data

*FFault determines how faults will be treated when programs are running.  If it is
*zero, all faults will be reported to Midas.  If FFault is nonzero, the kernel will
*send control through location 120 when a fault occurs and PARITY # 0 (faults with
*PARITY = 0 are breakpoints).
RV[FFAULT,66];

*Registers between 360 and 365 are used by the Midas overlays.  The following
*registers, used by WriteMI,  are also in this range.
RV[RADDR,65];
RV[RCNT,64];
RV[RW0,63];
RV[RW1,62];


*Constants for Recv and Send
MC[RecvByte,12];
MC[RecvWord,16];
MC[SendByte,21];
MC[SendWord,25];

IMRESERVE[,7501,11];       *space for Midas overlays (7500-7527)
IMRESERVE[,7513,15];

SET[CMDisp,7420];          *8-way dispatch on Midas command
SET[RWDisp, 7440];         *4-way dispatch on state bits of RWStat
```

```
*After loading kernel.mb, Midas starts it at 7000
Start:
        RTMP ← 1c, AT[7000];
        RTMP ← (RTMP) + (7000C),goto[KNotify];  *Notify Task 0, address 7001
KNotify:
        APC&APCTASK ← RTMP;
Kn1:
        RETURN;

        SETTASK[0];
*R definitions
RV[R0,0];
RV[R1,1];
RV[R2,2];
RV[R3,3];
RV[R4,4];
RV[R5,5];
RV[R6,6];
RV[R7,7];
RV[R10,10];
RV[R11,11];
RV[R12,12];
RV[R13,13];
RV[R14,14];
RV[R15,15];
RV[R16,16];
RV[R17,17];


*Clear R0-R17 to avoid R parity errors later
        R1 ← 0c, AT[7001];
        R2 ← 0c;
        R3 ← 0c;
        R4 ← 0c;
        R5 ← 0c;
        R6 ← 0c;
        R7 ← 0c;
        R10 ← 0c;
        R11 ← 0c;
        R12 ← 0c;
        R13 ← 0c;
        R14 ← 0c;
        R15 ← 0c;
        R16 ← 0c;
        R17 ← 0c;
*Clear R20-R377 using Stkp
        R0 ← 20c;
RClear:
        Stkp ← R0;
        Stack ← 0c;
        lu ← (R0) xor (377c);
        R0 ← (R0) + 1, goto[RClear,ALU#0];
        R0 ← 5c;            *Notify task 17, location 7005
        R0 ← (R0) + (177000c);
        APC&APCTASK ← R0, goto[Kn1];

        SETTASK[17];

        RTMP ← (400C), AT[7005];         *set Printer idle, don't drive bus
        Printer ← RTMP;

        RTMP ← (100000C);
ClrTimers:
        LOADTIMER[RTMP];          *Clear out all Timers
        RESETMEMERRS;   *Clear any pending memory errors
        FFAULT ← 40000c;          *Initialize so that Midas takes faults
        RTMP ← (RTMP) + 1;
        LU ← (RTMP) AND (17C);  *there are 16d timers
        REFR ← (0C), DBLGOTO[InitDone, ClrTimers, ALU=0];
InitDone:
        LU ← TIMER;     *Set up the Refresh timer
        RTMP ← (50000C);
        RTMP ← (RTMP) OR (277C);         *simple timer,value 11d,slot 17b
        LoadTimer[RTMP];

*Notify task 16, address 7030 to set up timer task
        RTMP ← (167000C);
        RTMP ← (RTMP) OR (30C), goto[KNotify];

        Call[TimerInitDone], AT[7030]; *Set TPC[16] to TimerTask

*The simple timer task assumes slot 17 expired, since all others were cleared.
TimerTask:
        Refresh[REFR];
        lu ← Timer;     *read timer to clear the wakeup
        REFR ← (REFR) + (20c);
        RTMP ← (50000C);          *build timer constant
        RTMP ← (RTMP) OR (277C);         *simple timer,value 11d,slot 17b
        AddToTimer[RTMP];
CheckStop:
        T ← Printer;
        RTMP ← T;
        LU ← (RTMP) AND (10000C) ;
        GOTO[MidasStop,ALU#0] ;
TimerRet:       RETURN;
MidasStop:
        LU ← T, goto[MidasStop],SetFault, AT[7003]; *Midas recognizes a mouse halt as
*a task 16 breakpoint that was not set by the user.  It continues from (absolute) MidasStop+1
MidasRestart:   return, AT[7004];
```

```
*return to task 17, address 7400
TimerInitDone:
        RIMP ← (177400C), goto[KNotify];
```

```
*Page Zero stuff
*We put the instruction for BufferRefill here..
x377x:  gotop[x377x], at[377];  *dummy instruction
        loadpage[0], goto[x377x], at[0]; *Emulator buffer refill code is on page 0
        T ← APCTASK&APC, AT[1]; *Fault entry.  Save APC first, then the other volatile regs.
        RXAPC ← T, AT[100];
        T ← GETRSPEC[147], AT[101];      *ctask, ncia
        RXCTASK ← T, AT[102];
        T ← (GETRSPEC[103]) xor (377c), AT[103];        *sstkp, stkp (stkp is read complemented)
        RXSTK ← T, AT[104];
        RTMP ← 20c, AT[105]; *Set stkp to 20 in case there was a stack overflow pending
        Stkp ← RTMP, AT[106];
        T ← (GETRSPEC[107]) xnor (0c), AT[107]; *aluresult, saluf (both read complemented)
        RXALU ← T, AT[110];
        T ← GETRSPEC[157], LOADPAGE[0], AT[111]; *page, parity, bootreason
        RXPPB ← T, RESETERRORS, AT[112];
*Notify Task 17, address 7505 (breakpoint communication)
        RTMP ← (177400C), AT[113];
        RTMP ← (RTMP) OR (105C), AT[114];
        APC&APCTASK ← RTMP, AT[115];
        RETURN, AT[116];
*The go overlay will send control to UserFault (120) if PARITY # 0 and FFAULT<0
UserFault:
        loadpage[17], at[120]; *User may overwrite these instructions if desired
        gotop[MidasFault], at[117];
```

```
*The following is the page 17 portion of the kernel.  We get here
*after setting up the timer task.
        RDATA ← 40000C, AT[7400];       *send #100 to Midas
        RWSTAT ← SendByte, CALL[Send];

NextCom:
        RWSTAT ← RecvByte, CALL[Recv], AT[7404];
        Dispatch[RDATA,15,3];
        DISP[DoOverlay];

DoOverlay:
        GOTO[OverlayArea], AT[CMDisp,0]; *Midas overlay
OverlayArea:
        return, AT[7500];        *placeholder for overlay
MidasFault:
        return, AT[7512];        *placeholder for fault in Midas go overlay

WriteMI:
        RWSTAT ← RecvWord, CALL[Recv], AT[CMDisp,2];     *Write Control Store
        RADDR ← T;
        RWSTAT ← RecvByte, CALL[Recv];  *Get Count (byte)
        RCNT ← T;
WriteMILoop:
        NOP;
        RWSTAT ← RecvWord, CALL[Recv];  *Get Data 0 (word)
        RW0 ← T;
        RWSTAT ← RecvWord, CALL[Recv];  *Get Data 1 (word)
        RW1 ← T;
        RWSTAT ← RecvByte, CALL[Recv];  *Get Data 2 (byte)
        LU ← RW0;       *T has data 2
        APC&APCTASK ← RADDR;
        WRITECS0&2;
        LU ← RW1, AT[CMDisp,12]; *force writecs to have JA.7 = 0
        APC&APCTASK ← RADDR;
        WRITECS1;
        RADDR ← (RADDR) + 1, AT[CMDisp,14]; *force writecs to have JA.7 = 0
        RCNT ← (RCNT) -1;
        GOTO[WriteMILoop, ALU#0];
        GOTO[NextCom];

*Read a single R register.  Midas will use an overlay to read RM 0 and RM 10 - RM 17,
*to avoid generating stack overflow.
ReadR:
        RWSTAT ← RecvByte, CALL[Recv], AT[CMDisp,4];     *Get Address
        STKP ← RDATA;
        nop;
        T ← STACK;
        RDATA ← T;
        RWStat ← SendWord, Call[Send];
        GOTO[NextCom];

*Write a single R register.  Midas will use an overlay to write RM 0 and RM 10 - RM 17,
*to avoid generating stack overflow.
WriteR:
        RWSTAT ← RecvByte, CALL[Recv], AT[CMDisp,6];     *Get Address
        STKP ← RDATA;
        RWSTAT ← RecvWord, CALL[Recv];  *Get (word) data
        STACK ← T, GOTO[NextCom];
```

```
*SUBROUTINES Send and Receive communicate with Midas.

Send:   T ← rsh[RDATA,10], AT[7460];     *get msbyte (location 7460 is known to overlays)
        RTMP ← T, goto[RPRT];    *will get WrStrb on
Recv:   RDATA ← Zero, AT[7464]; *location 7464 is known to overlays
RW:     Refresh[REFR];  *Refresh the memory
        RTMP ← 30c;
Dlyloop:        RTMP ← (RTMP) -1, goto[Dlyloop,R>=0];
        REFR ← (REFR) + (20C);
        T ← Printer; *Get Printer data
        RTMP ← T;
        T ← Printer; *Insist that the printer yield the same data three times.
        LU ← (RTMP)-(T);
        T← Printer, Goto[.+2,ALU=0];
        Goto[RW];
        LU ← (RTMP)-(T);
        Goto[.+2,ALU=0];
        Goto[RW];
        T ← LDF[RTMP,0,2]; *Get strobe/ack bits
        LU ← (LDF[RWSTAT,16,2]) xor (T);         *Compare to desired bits
        T ← RTMP, GOTO[RW,ALU#0];        *if reached, clear all bits
        RTMP ← 400C;
        Printer ← RTMP, RTMP ← T;        *restore RTMP
        Dispatch[RWSTAT,13,2];  *dispatch on state bits of rwstat
        LU ← (RWSTAT) AND (4C), DISP[ReadStrobeOff];     *setup byte/word

ReadStrobeOff:
        USECTASK, GOTO[ReadMore, ALU#0], AT[RWDisp,0];  *Get Another byte if word set
        T ← RDATA, RETURN;
ReadMore:
        RWSTAT ← RecvByte, GOTO[RW];     *Go get another byte

ReadStrobeOn:
        T ← RHMask[RTMP], AT[RWDisp,1]; *Get Data Byte
        RDATA ← (LSH[RDATA,10]) OR (T); *Merge Byte
        RWSTAT ← (RWSTAT) AND (4C);      *State←0, Look for RDStrb off, retain byte/word
        RTMP ← (100400C);        *Set RdAck
RPRT:   Printer ← RTMP, GOTO[RW];


*Here on Write Ack On - state ← 3
        RWSTAT ← (RWSTAT) XOR (11C), GOTO[RW], AT[RWDisp,2];

*Here on Write Ack Off
        RDATA ← LSH[RDATA,10], goto[SendMore, ALU#0], AT[RWDisp,3];
        lu ← Zero, goto[ReadStrobeOff]; *must do non-tasking return
SendMore:       RWSTAT ← SendByte, goto[Send];

END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        title[key];

* last modified by Johnsson, August 17, 1979  12:27 PM

* KeyBoard Translation Table from D0's to ALTO's
* Entry in the table is bitnumber*8+wordnumber

m@[byte,DATA[(LH[LSHIFT[#1,10],#2],RH[LSHIFT[#3,10],#4],
  RSEL2@[pp[#1,#2,#3,#4]]
   ,at[byteloc])]
    SET[byteloc,ADD[byteloc,1]]];
m@[pp,set[ppx,xor@[1,#1,#2,#3,#4]]set[ppx,xor@[ppx,rshift[ppx,4]]]
  set[ppx,xor@[ppx,rshift[ppx,2]]]set[ppx,xor@[ppx,rshift[ppx,1]]]
   set[ppx,and@[1,ppx]]ppx];

set[byteloc,keytable];

byte[177,177,177,177];   *00
byte[177,177,177,177];
byte[177,177,177,177];
byte[177,177,177,177];

                         *04:
byte[005,150,115,105];   * D1, T10(\), T9, T8.
byte[075,171,065,163];   * T7, R6(BW), T6, L12(FL4).
byte[172,160,055,045];   * L9(FL3), L6(LF), T5, T4.
byte[035,025,012,177];   * T3, T2, T1(esc), .

                         *10:
byte[164,177,125,144];   * R4, , R1, R2.
byte[004,173,177,024];   * R5, R3(FR5), , L10.
byte[034,044,054,177];   * L7, L4, L1.
byte[177,064,177,177];   * , A9, , .

                         *14:
byte[154,074,014,155];   * R7, R10, R11, R8.
byte[161,153,113,104];   * R9(FR4), R12(swat), A7(space), L11.
byte[114,124,134,162];   * L8, L5, L2, L3(DEL).
byte[042,177,177,177];   * A8(CTL), , , .

                         *20:
byte[177,177,143,140];   * , , A6(shift-R), /.
byte[122,131,073,063];   * ., (comma), m, n.
byte[072,070,052,101];   * b, v, c, x.
byte[102,177,135,177];   * z, , 47, .

                         *24:
byte[142,152,141,132];   * A4(return), 46(←), (quote), :.
byte[121,110,062,043];   * l, k, j, h.
byte[023,032,050,041];   * g, f, d, s.
byte[051,177,103,112];   * a, , A3(lock), A5(shift-L).

                         *30:
byte[145,151,123,130];   * A10, 45(]) , 42([), p.
byte[111,071,060,033];   * o, i, u, y.
byte[013,003,030,021];   * t, r, e, w.
byte[031,022,177,174];   * q, A1(tab), , D2.

                         *34:
byte[170,133,120,100];   * A2(bs), =, -, 0
byte[061,053,040,020];   * 9, 8, 7, 6.
byte[000,010,001,011];   * 5, 4, 3, 2.
byte[002,015,177,177];   * 1, 48, , .

        end[key];
```

```
TITLE[uiDEFS];
*last edit by Chang on September 10, 1979  4:45 PM, move Timer's regs
* edit by CPT  December 22, 1978  3:18 AM
* edit by Sandman  March 23, 1979  3:02 PM
* edit by Jarvis  August 10, 1979  11:35 AM

*UIDEFS.MC -definitions for IUTFP revision I

SET[uiUTFPBASEADDR,LSHIFT[uiUTFPPAGE,10]]; *FIRST ADDRESS OF UTFP PAGE


*REGISTERS AND CONSTANTS USED BY UTFP TASK
SETTASK[uiUTFPTASK];

SET[uiREADSTATREG,1];
SET[uiDBREG,1];
SET[uiCREG,2];
SET[uiHEBUF,3];
SET[uiCXREG,4];
SET[uiHTAB,5];
SET[uiBPREG,6];
SET[uiCURSM,7];
SET[uiDBADDR,ADD[lshift[uiUTFPTASK,4],1]];

MC[BlkBkgndBit,100];

RV[uiDWA,0]; *bit map base register
RV[uiDWA1,1];
RV[uiTEMP,2]; *must be even/odd pair, see uiDCBDONE for PFETCH2
RV[uiTEMP1,3];
RV[uiMPSTATUS,4]; *keyboard decode state, bits 10-12: count, 13-17: part
RV[uiMOUSEDELXY,5]; *BITS 0-5: XDELTA, 10-15: YDELTA
RV[uiTMSG,6]; *INCOMING PARTIAL MESSAGE
RV[uiXMSG,7]; *MESSAGE HELD FOR POSTING BY VSYNC

RV[uiLINK,10]; *DISPLAY CONTROL BLOCK WORD 0
RV[uiNWRDS,11]; *DISPLAY CONTROL BLOCK WORD 1
RV[uiDBA,12]; *DISPLAY CONTROL BLOCK WORD 2
RV[uiHEPAT,12]; *Used during initialization only
RV[uiSLC,13]; *DISPLAY CONTROL BLOCK WORD 3
RV[uiHEADDR,13]; *Used during initialization only
RV[uiQTEMP,10]; *Used for Store4 to post mouse buttons (uiVS4)
RV[uiQTEMP1,11];
RV[uiQTEMP2,12];
RV[uiQTEMP3,13];

RV[uiBUFPTR,14];
RV[uiHECNT,14]; *Used during initialization only
RV[uiCRWORD,15]; *IMAGE OF HARDWARE CONTROL REGISTER
RV[uiVSCOUNT,16]; *Count of lines per field.
RV[uiHELINK,16]; *Used during initialization only
RV[uiLINESPERFIELD,17]; *404 = 624b lines per field
mc[lpflo,224];
mc[lpfhi,400];

RV[uiCC0,10];    * used during initialization only
RV[uiCC1,11];    * used during initialization only
RV[uiCC2,12];    * used during initialization only
RV[uiCC3,13];    * used during initialization only

* RV[RTCLOW,25]; *Must be the same as timer's
RV[RTCLOW,55]; *Must be the same as timer's

RV[uiCX,30]; *Cursor X
RV[uiCY,31]; *Cursor Y
RV[uiCNT,32];
RV[uiBASE,34]; *BASE REGISTER PAIR
RV[uiBASE1,35];
RV[uiNBUFPTR,36];
RV[uiBUTTONS,37];
```

```
*       horizontal event RAM
*count her[1:9] and patterns her[12:15] each bit represents 4 pixels of scan
*       her[12] CLR
*       her[13] HS
*       her[14] Blank
*       her[15] Half Line Clock
mc[her0,   400]; mc[her0p, 203]; *   6,           Blank, Half Line   6
mc[her1, 13000]; mc[her1p, 101]; * 131,                  Half Line  89
mc[her2,     0]; mc[her2p, 200]; *   2                               2
mc[her3, 25000]; mc[her3p,   1]; * 250,                  Half Line 168
mc[her4,  1000]; mc[her4p, 203]; *  12,           Blank, Half Line  10
mc[her5,     0]; mc[her5p, 206]; *   2,      HS, Blank               2
mc[her6, 12000]; mc[her6p, 107]; * 121,      HS, Blank, Half Line  81
mc[her7,     0]; mc[her7p, 117]; *   1, CLR, HS, Blank, Half Line   1
mc[her8,     0]; mc[her8p, 103]; *   1,           Blank, Half Line   1
mc[her9, 23000]; mc[her9p,  12]; * 230, CLR,      Blank            152
************************** total 1000 ************************** 512
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTOTB;
insert[GlobalDefs];
insert[LFDefs];
	title[UIInit];
*last edit by Johnsson  April 7, 1979  12:29 PM
*last edit by Jarvis  June 13, 1979  9:55 AM

*Initialization for IUTFP

SETTASK[uiUTFPTASK];
ONPAGE[DisplayInitPage];


displayinit:  uiHEADDR ← (ZERO), AT[DisplayInitLoc];
	OUTPUT[uiHEADDR,uiCREG]; *CLEAR THE CONTROL REGISTER

	uiHEPAT ← her0 ; *LOAD THE HORIZONTAL EVENT RAM
	uiHEPAT ← (uiHEPAT)or(her0p), CALL[uiLOADHE];

	uiHEPAT ← (her1) ;
	uiHEPAT ← (uiHEPAT) or (her1p), CALL[uiLOADHE];

	uiHEPAT ← (her2) ;
	uiHEPAT ← (uiHEPAT) or (her2p), CALL[uiLOADHE];


	uiHEPAT ← (her3) ;
	uiHEPAT ← (uiHEPAT) or (her3p), CALL[uiLOADHE];

	uiHEPAT ← (her4) ;
	uiHEPAT ← (uiHEPAT) or (her4p), CALL[uiLOADHE];


	uiHEPAT ← (her5) ;
	uiHEPAT ← (uiHEPAT) or (her5p), CALL[uiLOADHE];


	uiHEPAT ← (her6) ;
	uiHEPAT ← (uiHEPAT) or (her6p), CALL[uiLOADHE];

	uiHEPAT ← (her7) ;
	uiHEPAT ← (uiHEPAT) or (her7p), CALL[uiLOADHE];

	uiHEPAT ← (her8) ;
	uiHEPAT ← (uiHEPAT) or (her8p), CALL[uiLOADHE];

	uiHEPAT ← (her9) ;
	uiHEPAT ← (uiHEPAT) or (her9p), CALL[uiLOADHE];

uiHELOADED:
	uiBASE ← zero;
	uiBASE1 ← zero ;
	uiLINESPERFIELD ← lpflo ;
	uiLINESPERFIELD ← (uiLINESPERFIELD) or (lpfhi) ;

*set keyboard words to -1 (key up)
	uiTEMP ← 177000C;
	T ← uiTEMP ← (uiTEMP) or (30C); * 177030
	uiCC0 ← (ZERO)-1;
	uiCC1 ← (ZERO)-1;
	uiCC2 ← (ZERO)-1;
	uiCC3 ← (ZERO)-1;
	PSTORE4[uiBASE,uiCC0];		* mouse,keyset,etc.
	T ← uiTEMP ← (uiTEMP)+(4c);	* 177034
	PSTORE4[uiBASE,uiCC0];		* keyboard[0:3].
	T ← uiTEMP ← (uiTEMP)+(4c);	* 177040
	PSTORE4[uiBASE,uiCC0];		* keyboard[4:7].

	uiCRWORD ← (220C); *ALLOW WAKEUPS, CDIAG←0
	OUTPUT[uiCRWORD,uiCREG]; *ALLOW WAKEUPS
	uiTEMP ← 377C;
	OUTPUT[uiTEMP,uiHTAB]; *load the HTAB counter with 377
	uiTMSG ← (ZERO);
	uiMPSTATUS ← T ← ZERO, call[uiFINHE];
	uiLINK ← T, loadpage[uiutfppage]; *First wakeup comes here
	uiBUFPTR ←T← 377C,GOTOp[uiCSDONE];

*SUBROUTINE TO LOAD THE HORIZONTAL EVENT RAM
*(ADDRESSED VIA CXREG)
uiLOADHE:	T ← LDF[uiHEPAT,1,11];
	uiHECNT ← (T);
uiHELOADLOOP: uiHECNT ← (uiHECNT)-1;
	GOTO[uiFINHE,ALU<0], usectask;
	OUTPUT[uiHEADDR, uiCXREG];
	uiHEADDR ← (uiHEADDR)+1;
	OUTPUT[uiHEPAT, uiHEBUF],goto[uiHELOADLOOP];
uiFINHE:	RETURN;

	end[uiinit];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        title[key];

* KeyBoard Translation Table from D0's to ALTO's

m@[byte,DATA[(LH[LSHIFT[#1,10],#2],RH[LSHIFT[#3,10],#4],
  RSEL2@[pp[#1,#2,#3,#4]]
   ,at[byteloc])]
    SET[byteloc,ADD[byteloc,1]]];
m@[pp,set[ppx,xor@[1,#1,#2,#3,#4]]set[ppx,xor@[ppx,rshift[ppx,4]]]
  set[ppx,xor@[ppx,rshift[ppx,2]]]set[ppx,xor@[ppx,rshift[ppx,1]]]
   set[ppx,and@[1,ppx]]ppx];

set[byteloc,keytable];

byte[177,177,177,177];  *00
byte[177,177,177,177];
byte[177,177,177,177];
byte[177,177,177,177];

byte[5,125,115,105];     *20: D1, T10, T9, T8.
byte[75,171,65,163];     * T7, BW, T6, FL4.
byte[172,160,55,45];     * FL3, LF, T5, T4.
byte[35,25,15,177];      * T3, T2, T1, .

byte[151,177,153,150];   *40: ], , (swat), \.
byte[152,14,177,24];     * ←, R3, , L10.
byte[34,44,54,177];      * L7, L4, L1.
byte[177,64,177,177];    * , A9.

byte[152,74,173,161];    *60: ←, R10, FR5, FR4.
byte[177,173,113,104];   * , FR5, (space), L11.
byte[114,124,134,162];   * L8, L5, L2, DEL.
byte[42,177,177,177];    * CTL.

byte[177,177,143,140];   *100:, , shift-R, /.
byte[122,131,73,63];     * ., (comma), m, n.
byte[72,70,52,101];      * b, v, c, x.
byte[102,177,42,177];    * z, , (ctl), .

byte[142,154,141,132];   *120: return, 46, (quote), :.
byte[121,110,62,43];     * l, k, j, h.
byte[23,32,50,41];       * g, f, d, s.
byte[51,177,103,112];    * a, , lock, shift-L.

byte[171,151,123,130];   *140: BW, ] , [, p.
byte[111,71,60,33];      * o, i, u, y.
byte[13, 3,30,21];       * t, r, e, w.
byte[31,22,177,174];     * q, tab, , D2.

byte[170,133,120,100];   *160: bs, +, -, 0
byte[61,53,40,20];       * 9, 8, 7, 6.
byte[ 0, 10, 1, 11];     * 5, 4, 3, 2.
byte[ 2, 12,177,177];    * 1, esc.

        end[key];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[LFDefs];
TITLE[extended-address-UITASK];

* Last modified by Chang, September 10, 1979  4:56 PM, move Timer's Regs';
* modified by Johnsson, April 7, 1979  12:29 PM;
* modified by Jarvis, June 27, 1979  4:00 PM
SETTASK[uiUTFPTASK];
ONPAGE[uiUTFPPAGE];

SET[uiPART,ADD[uiUTFPBASEADDR,20]]; *backchannel message dispatch
SET[uiSHCUR,ADD[uiUTFPBASEADDR,60]]; *cursor shift

* uiMPSTATUS holds keyboard process state information
* The process intializes the count field with the negative of the number of
* bits in the field.  As each bit comes in, the process increments the count.
* When the count reaches zero, the carry out of the count increments the
* state.  The field sizes and their associated state are:
* state size
*   0         idle
*   1      4  x mouse delta, twos complement
*   2      4  y mouse delta
*   3      3  mouse buttons (and status are lumped together in single field)
*          4  status (video, VS, power supply normal, key data follows)
*   4     10  key data (7 bits of key, 1 bit of up/down)
*   5         post key data
mc[keyStart, 74];  * initial state, count=-4, state=1

* The mouse resolves 200 pixels/inch.  35 inches/sec is the maximum speed
* required for tracking the mouse.  We update the mouse position once every
* field, 80 times a second.  Therefore tracking the mouse at maximum velocity
* requires a field large enough hold 88 counts.  Adding 1 bit for sign
* mandates an 8 bit field for x and another 8 bit field for y.  Note that this
* uses 7 bit fields.

* check for message from key board
* calling sequence:  lu← uiMPSTATUS, call[keyCheck];
* does not task!!!!
keyCheck: uiMPSTATUS← (uiMPSTATUS)+1, goto[keyContinue, alu#0];
        skip[IO atten], use CTask;  *state=1, count=-4;
          return, uiMPSTATUS← 0C;  * reset to state=count=0
        return, uiMPSTATUS← keyStart;

keyContinue: uiTMSG← rsh[uiTMSG, 1], skip[no atten];
          uiTMSG← (uiTMSG) OR (100000C);
        lu← ldf[uiMPSTATUS, 13, 5];
        skip[alu=0], use CTask;
          return;  * more bits to go in this field

* have accumulated all bits for this field, dispatch on state
        dispatch[uiMPSTATUS, 10, 3];
        disp[.+1];             *finish this field *negative count for next field
uiMPSTATUS← (uiMPSTATUS) or (34c), goto[keyXY], at[uiPART, 2]; *y
uiMPSTATUS← (uiMPSTATUS) or (31c), goto[keyXY], at[uiPART, 3]; *buttons/status
uiMPSTATUS← (uiMPSTATUS) or (30c), goto[keyBS], at[uiPART, 4]; *key
t← rsh[uiTMSG, 10],            goto[keyStroke], at[uiPART, 5]; *reset

keyXY:  uiTMSG← rsh[uiTMSG, 14], skip[r>=0];
          uiTMSG← (uiTMSG) OR (170C); *negative number, extend sign
        lu← (uiMPSTATUS) AND (40C); *kludgy test for x or y
        t← lsh[uiTMSG, 11], skip[ALU=0];
          t← uiTMSG; *this is y
        uiMOUSEDELXY← (uiMOUSEDELXY)+t;
        uiMOUSEDELXY← (uiMOUSEDELXY) AND NOT (400C), goto[keyNoTaskFieldDone];

keyBS:  t← rsh[uiTMSG, 7];
        uiBUTTONS← t;
        lu← uiTMSG, use C Task, dblgoto[keyFieldDone, keyIdle, r<0];

keyStroke: uiXMSG← (lsh[uiXMSG, 10]) or t;
keyIdle: uiMPSTATUS← 0C;
keyNoTaskFieldDone: use CTask;
keyFieldDone: uiTMSG← 0c, return;
```

```
*Do horizontal processing.  We know that the controller needs data.
xpreVS0:        lu← uiMPSTATUS, call[keyCheck];
preVS0: T← 2C, call[uiCheckCursor];

uiVS0:  OUTPUT[uiNBUFPTR,uiBPREG]; *precomputed uiNBUFPTR to hardware.
        T ← (uiDBA) AND NOT (17C);
        uiDWA ← T;
*Start the first IOFETCH
uiDWT:  IOFETCH16[uiBASE,uiDBADDR];

*Calculate the read buffer pointer, the count, and next line's.DBA
*in the shadow of the first IOFETCH (If there is to be more than one).
        T ← uiBUFPTR ← 377C;
        uiNBUFPTR ← (uiNBUFPTR)-(357C); *From here on, uiNBUFPTR is used
*for the count (-(NWRDS + (ADDRESS and 17B)))
        T←RHMASK[uiNWRDS];
        uiBUFPTR ← (uiBUFPTR)-(T); *377- number of words for the display
        uiBUFPTR ← (uiBUFPTR) OR (100000C); *Wakeup disable bit
        T← (RHMASK[uiNWRDS])+(T); * T ← 2*NWRDS
        uiDBA←(uiDBA)+(T); *uiDBA is now set up for the next scan line
        uiDWA←T←(uiDWA)+(20C),CALL[uiDWT1];

*Loop for second through Nth IOFETCH.
uiDWT1: uiNBUFPTR←(uiNBUFPTR)+(20C),GOTO[uiBUFD2,R>=0];
        IOFETCH16[uiBASE,uiDBADDR],GOTO[uiBUFD2X,ALU>=0];
        uiDWA←T←(uiDWA)+(20C), RETURN;

uiBUFD2: uiVSCOUNT ← (uiVSCOUNT)-1,DBLGOTO[uiENDFIELD,uiCONT,R<0]; *check for field done
uiBUFD2X: uiVSCOUNT ← (uiVSCOUNT)-1,DBLGOTO[uiENDFIELD,uiCONT,R<0];
uiCONT: uiSLC←(uiSLC)-1,DBLGOTO[uiDCBDONE,uiMDCB2,R<0];


*Calculate the next  line's uiNBUFPTR
*in the shadow of the last IOFETCH16
uiMDCB2: uiNBUFPTR ← 377C;
        T←LDF[uiDBA,14,4];
        uiNBUFPTR ← (uiNBUFPTR)-(T);
        T←RHMASK[uiNWRDS];
        uiNBUFPTR ← (uiNBUFPTR)-(T);
        OUTPUT[uiBUFPTR,uiBPREG], goto[xpreVS0];

*The DCB is finished.
uiDCBDONE:      lu← uiMPSTATUS, call[keyCheck];
        T←(uiLINK);
        uiBASE1 ← 0C, GOTO[uiGetNextDCB,ALU#0];
*The DCB chain is exhausted.
        OUTPUT[uiBUFPTR,uiBPREG]; *Send read BUFPTR to the hardware
        T ← 2C, call[uiCheckCursor]; *does TASK return
        goto[uiVS1];


uiGetNextDCB:
        uiBASE ← T; *T contains LINK. Set base register to point to next DCB
        OUTPUT[uiBUFPTR,uiBPREG]; *Send read BUFPTR to the hardware
        uiNBUFPTR ← 377C; *Init for later
        nop;*two instr after output
        PFETCH2[uiBASE,uiDBA,2]; *Fetch DBA,SLC
        uiBUFPTR ← 377C; *Init for later
*Check for long pointer addressing
        PFETCH2[uiBASE,uiLINK,0]; *Fetch Link,NWRDS
        LU←uiSLC,goto[uiLong,R<0];

*Short Pointer
        T ← uiDBA;
        uiBASE ← T,goto[uiEvenOdd];


*Long Pointer
uiLong:
        PFETCH2[uiBASE,uiBASE,4]; *fetch directly into the base register
        uiSLC ← (uiSLC) AND NOT (100000C);  *clear the sign bit


*Bias uiDCB.SLC by -2.  Note that if uiDCB.SLC = 0 OR 1, at least one
*scan line will be displayed.

uiEvenOdd: LU ← LDF[uiCRWORD,17,1];     *Check EvenField
        uiSLC ← (uiSLC)-(2C),GOTO[uiDBAOK,ALU#0];

uiDBABAD: T← RHMASK[uiNWRDS];
        uiBASE ← (uiBASE)+(T);
uiDBAOK: T←LDF[uiBASE,14,4]; *Set up NBUFPTR for the next scan
        uiNBUFPTR ← (uiNBUFPTR)-(T); *uiNBUFPTR ← 377C earlier
        T←RHMASK[uiNWRDS];
        uiNBUFPTR←(uiNBUFPTR)-(T);

*Now fix up the base register so that it is hex aligned and DBA contains the residue

        T ← (uiBASE) and (17C);
        uiDBA ← T;
        uiBASE ← (uiBASE) and not (17C);
*fix up the high half of the base register
        T ← lsh[uiBASE1,10];
        uiBASE1 ← (RHMASK[uiBASE1])+(T)+1;

        T← 2C, call[uiCheckCursor];  *returns to VS2

*We have just picked up a new DCB.  We must output HTAB,
*then go to normal state 0 processing.
```

```
uiVS2:  T ← LDF[uiNWRDS,2,6]; *calculate HTAB
        LU←LDF[uiNWRDS,1,1]; *black background bit
        uiBUFPTR ← (uiBUFPTR) - (T)-1,GOTO[.+2,ALU#0]; *uiBUFPTR ← 377C earlier
        uiBUFPTR ← (uiBUFPTR) AND NOT (200C);
        OUTPUT[uiBUFPTR,uiHTAB],goto[uiVS0]; *send it

uiENDFIELD:     uiBASE1 ← 0C;
        uiCRWORD ← (uiCRWORD) xor (3C), goto[uiFD1];
```

```
*We have displayed the entire chain and the field is not done.
*Wait for end of field.
xpreVS1:        T← 2C, call[uiCheckCursor];
preVS1: call[uiWAKEOFF];
uiVS1: lu← uiMPSTATUS, call[keyCheck];
        uiVSCOUNT ← (uiVSCOUNT)-1, GOTO[uiFIELDDONE,R<0];
        goto[xpreVS1];


*Turn off wakeallow and return
uiWAKEOFF:  uiCRWORD ← (uiCRWORD) AND NOT (200C); *AllowWake ← 0
        OUTPUT[uiCRWORD,uiCREG];
        uiCRWORD ← (uiCRWORD) OR (200C);
        uiBUFPTR ← 377C; *set up for next run.
        OUTPUT[uiCRWORD,uiCREG], goto[Outputwait];

*Check for cursor visible.  Enter with T=2.
uiCheckCursor:
        uiCY← (uiCY)-(T), GOTO[uiCURRET, R>=0];
        uiCX ← (uiCX)+(1000C);
        LU ← LDF[uiCX,3,1];
        GOTO[uiSENDCX,ALU=0];
        uiCX ← 5C;        *finished displaying the cursor
        uiCY ← 10000C;
uiSENDCX:
        OUTPUT[uiCX,uiCXREG];
Outputwait: nop;
uiCURRET:        return;

*The field is finished.  Make Vsync pulse.
uiFIELDDONE:
        uiCRWORD ← (uiCRWORD) XOR (3C);  *complement field, set PreVS.
uiFD1:  uiBUFPTR ← 200C;
 *send black background so that hsync won't be screwed up
        OUTPUT[uiBUFPTR,uiHTAB], call[uiWAKEOFF]; *returns to uiVS3
```

```
*We are in the first scan line of a vertical sync pulse.
*Post the mouse COORDINATES to core.
uiVS3: uiBASE ← 0C;
        T← (uiBUFPTR)+(25C); *424b AND 425b = Mouse x and y.
        PFETCH2[uiBASE,uiDBA]; *uiDBA and uiSLC are used as temps.
        lu← uiMPSTATUS, call[keyCheck];
        uiMOUSEDELXY ← LCY[uiMOUSEDELXY,11],DBLGOTO[uiSEM1,uiNSEM1,R<0];

uiSEM1: T ← 177C,GOTO[uiSEM1FIN];
uiNSEM1: T ← (ZERO)-1;
uiSEM1FIN: T← (LDF[uiMOUSEDELXY,7,7]) XNOR (T);
        uiDBA ← (uiDBA)+(T);

        T←uiMOUSEDELXY←LDF[uiMOUSEDELXY,0,7],DBLGOTO[uiSEM2,uiNSEM2,R<0];

uiSEM2: T ← (uiMOUSEDELXY) XNOR (177C);
uiNSEM2: uiSLC ← (uiSLC)+(T);
        T← (uiBUFPTR)+(25C);
        PSTORE2[uiBASE,uiDBA]; *Restore coordinates
        uiMOUSEDELXY ← (ZERO), call[uiWAKEOFF]; *returns to uiVS4
```

```
*Post the mouse BUTTONS
uiVS4: uiBASE←177000C; *NOTE modification of uiBASE
        uiBASE←(uiBASE)+(30C); *Fetch 177030
        PFETCH1[uiBASE,uiQTEMP,0]; *uiQTEMP overlays LINK, NWRDS, DBA, SLC.
        lu← uiMPSTATUS, call[keyCheck];

*convert UTFP mouse button order into ALTO order
*On the UTFP, the sequence for the buttons (BUTTONS[13:15]) is right,
*middle,left, and 1's mean buttons depressed.
*On the ALTO, 177030[15:17] correspond to left,right,middle, and 1's
*in memory mean button NOT depressed.

        t← ldf[uiBUTTONS,13,1]; *right button
        uiTEMP ← t;
        t ← ldf[uiBUTTONS,14,1]; *middle button
        uiTEMP ← (lsh[uiTEMP,1]) or (t);
        t ← (uiBUTTONS) and (4C); *left button
        t ← (uiTEMP) or (t);
        uiQTEMP ← (uiQTEMP) or (7C);
        t ← uiQTEMP ← (uiQTEMP) xor (t);
        * ignore uiQTEMP1 and uiQTEMP2
        uiQTEMP3 ← t;
        nop;     * wait for write of register
        PSTORE4[uiBASE,uiQTEMP,0];
        uiBASE ← ZERO; *reset base register


*Fetch new dcb chain header and interrupt mask.
        T ← (uiBUFPTR)+(21C); *T←420, Since uiBUFPTR=377.
        PFETCH2[uiBASE,uiLINK]; *Get new dcb header from 420, intmask from 421.


*check for realtime clock update
*save STKP
*       uiTEMP ← 325C; *Point to RTCLOW
        uiTEMP ← 355C; *Point to RTCLOW
        T← nSTKP;
        STKP ← uiTEMP, uiTEMP ← T,NoRegILockOK;
        uiTEMP ← (uiTEMP) XOR (377C); *STKP read inverted
        STACK ←(STACK) AND NOT (100000C),GOTO[uiNORTCOV,R>=0];
*must update RTC
        uiTEMP1←400C;
        T←(uiTEMP1)+(30C);
        PFETCH1[uiBASE,uiTEMP1];
        uiTEMP1 ← (uiTEMP1) + 1;
        PSTORE1[uiBASE,uiTEMP1];
*cause vertical field interrupt
uiNORTCOV:      STKP ← uiTEMP;
        loadpage[0];
        T ← uiNWRDS, callp[DoInt]; *interrupt mask fetched from 421 above

uiNOTMR:        call[uiWAKEOFF]; *Returns to uiVS5
```

```
*Post the keyboard

uiVS5:  lu ← LHMASK[uiXMSG], call[uiKPOST];
        lu ← LHMASK[uiXMSG], call[uiKPOST]; *do it again for other byte
        lu← uiMPSTATUS, call[keyCheck];
        uiBASE ← ZERO, GOTO[preVS6];

uiKPOST:
        uiTEMP ← KeyTableH,goto[.+2,alu#0]; *if no data, return right away
        uiXMSG ← lsh[uiXMSG,10],return; *shift to other keyboard char

        uiTEMP ← (uiTEMP) or (KeyTableL);
        t ← ldf[uiXMSG,1,5]; *Get word number (4 bytes per word)
        uiTEMP ← (uiTEMP) + (T); *Form final address
        t ← ldf[uiXMSG,6,1]; *set h2 to high/low word
        APC&APCTASK ← uiTEMP; *Address to read in Control Store
        READCS; *get the word
        t ← CSDATA, AT[uiUTFPBASEADDR,300]; *must be at an even location for READCS
        uiDBA ← t;
        lu ← LDF[uiXMSG,7,1]; *low or high byte
        goto[.+2,alu#0], uiBASE ← 177000C;
        uiDBA ← RSH[uiDBA,10]; *Need upper byte
        uiBASE ← (uiBASE)+(34C); *uiBASE ← 177034C
        T←(LDF[uiDBA,15,3]); *Get word number
        PFETCH1[uiBASE,uiNWRDS]; *uiNWRDS is a temp - fetch Alto kbd word
        uiDBA ← LDF[uiDBA,11,4]; *Get bit number
        uiBASE ← (uiBASE)+(T); *fix base register for store
        uiTEMP ← T ← 100000C; *Do the function uiTEMP ← 100000 rshift uiDBA

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 15
        uiTEMP ← T ← RSH[uiTEMP,1]; *shift 1

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 14
        uiTEMP ← T ← RSH[uiTEMP,2]; *shift 2

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 13
        uiTEMP ← T ← RSH[uiTEMP,4]; *shift 4

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 12
        uiTEMP ← T ← RSH[uiTEMP,10]; *shift 8


uiKDD:  *test for key down (0) or up (1)
        uiXMSG ← lsh[uiXMSG,10], DBLGOTO[uiKDOWN,uiKUP,r>=0];

uiKDOWN: uiNWRDS ← (uiNWRDS) AND NOT (T), GOTO[uiKSTORE]; *key down, clear bit
uiKUP: uiNWRDS ← (uiNWRDS) OR (T); *key up, set bit
uiKSTORE: goto[uiCURRET],PSTORE1[uiBASE,uiNWRDS,0]; *store word

preVS6: uiBUFPTR ← 377C, call[uiWAKEOFF]; *returns to uiVS6
```

```
*Set up the cursor.
*Get cursor coordinates from 426b (X), and 427b (Y).
uiVS6: T  ← (uiBUFPTR)+(27C); *uiBUFPTR = 377 on entry.
        PFETCH2[uiBASE,uiCX]; *Cursor X,Y coordinates
        lu← uiMPSTATUS, call[keyCheck];
        uiCY ← (uiCY)+1;

*The cursor X counter is loaded by HSF from CXREG.  It is
*clocked by [(EdgeClock and SelCursM) or (NClk and (HSF or VS'))].

*The cursor hardware counts the X offset from end of horizontal synch.  We
*must fudge for interval between horizontal synch and blank.
        uiCX← (uiCX)+(40C);

*The cursor buffer is divided into 8 segments with 6 nibbles in a segment.
*The cursor itself is 16 bits (4 nibbles) wide but a segment must have 5
*nibbles to allow for a cursor not aligned on a nibble boundary.  The sixth
*nibble must be all zero.  The harware ors this nibble with all displayed
*nibbles following the cursor.  Loading CXREG with a 5 disables the cursor
*by oring this zero nibble from the cursor with display data all across the
*line.

*When the cursor is visible, -X is loaded into the CXREG in the
*scan line preceding the cursor, and this value is loaded into
*the cursor counter by HS.  When VS=0, the cursor counter is
*incremented by NClk, and when it becomes 0, the next 5 nibbles are
*sent to the display.

*Here, we want the cursor counter to address the cursor buffer so
*that we can load it one segment at time.  The cursor segment is CXREG[4:6].
*We start with segment 1, CXREG← 1000C, and during each of the
*next 8 scan line times we will send 5 bytes of cursor data followed
*by a sixth nibble of zero to the buffer, then increment the segment value in
*uiCXREG.

*uiVSCOUNT is used to hold the value to be loaded into uiCXREG.
        uiVSCOUNT ← 1000C; *Start load at segment 1.

*The even scan lines of the cursor will be displayed if CY is even
*and the field is even, or if CY is odd and the field is odd.
*Otherwise, the odd scan lines will be displayed.
        uiBUFPTR ← (uiBUFPTR)+(32C); *uiBUFPTR ← 431b
        T← LDF[uiCRWORD,17,1];  *adjust offset according to even/odd field
        T← (LDF[uiCY,17,1]) XOR (T);
        T← (uiBUFPTR)+T;        *starts at either 431b or 432b
        uiDBA ← T, goto[uiMORECSETUP];
```

```
*Load one segment of the cursor memory (5 nibbles plus one zero nibble).
*The address was set up during the previous scan line.

uiMORECSETUP: OUTPUT[uiVSCOUNT, uiCXREG], call[uiWAKEOFF];
        T← uiDBA;  *Pointer to cursor segment
        PFETCH1[uiBASE, uiTEMP]; *Fetch segment
        lu← uiMPSTATUS, call[keyCheck];
        uiDBA← (uiDBA)+(2C); *Increment pointer (by 2 due to interlace)
        DISPATCH[uiCX, 16, 2];  *Determine amount to shift word.
        DISP[.+1], uiTEMP1← 17C;
          T← 0C, GOTO[uiSHCDONE],                       AT[uiSHCUR,0];
          T← LSH[uiTEMP, 3],                            AT[uiSHCUR,1];
            uiTEMP← RSH[uiTEMP, 1], GOTO[uiSHCDONE];
          T←LSH[uiTEMP, 2],                             AT[uiSHCUR,2];
            uiTEMP← RSH[uiTEMP, 2], GOTO[uiSHCDONE];
          T← LSH[uiTEMP, 1],                            AT[uiSHCUR,3];
            uiTEMP← RSH[uiTEMP, 3], GOTO[uiSHCDONE];

uiSHCDONE: uiTEMP1← (uiTEMP1) AND (T);
        uiCNT← 2C;

uiSENDCLOOP: T← LDF[uiTEMP, 0, 4];  *Loop for first 4 bytes
        uiNWRDS← T; *uiNWRDS is a temporary, not used during VS.
        OUTPUT[uiNWRDS, uiCURSM];
        uiCNT← (uiCNT)-1, skip[R<0];
        uiTEMP← LSH[uiTEMP, 4], GOTO[uiSENDCLOOP];

        OUTPUT[uiTEMP1, uiCURSM]; *Send 5th nibble.
        LU← LDF[uiVSCOUNT, 3, 1];
        OUTPUT[uiTEMP, uiCURSM], skip[ALU#0]; *0 for 6th byte
        uiVSCOUNT← (uiVSCOUNT)+(1000C), GOTO[uiMORECSETUP];      *next segment

        T← RSH[uiCX, 2]; *cursor loaded, CX counts nibbles, not bits
        uiCX← (ZERO)-T; *And it is negated.
        LU← uiCRWORD, skip[R ODD]; *Test field
          uiCY← (uiCY)-(1C); *even field
uiCSDONE: uiVSCOUNT← 5C; *initialization code jumps in here
        OUTPUT[uiVSCOUNT, uiCXREG], call[uiWAKEOFF]; *disable cursor

**********************************************************
*How do I love thee?
*Let me count the ways.

*This is the best accounting that I can mannage for what goes on in the
*horizontal scans during vertical retrace

*uiVS3 mouseXY                       1
*uiVS4 buttons, RTC, field wakeup    1
*uiVS5 keyboard                      1
*uivs6 cursor setup                  1
*      load cursor                   8
*-----------------------------------
*      subtotal                      12

*The UTLF gives 404 or 405 visible lines for even or odd fields respectively
*(see uiEVX and uiODX).  In order achieve an 875 line scan the UTLF must idle
*21 lines.
          uiVSCOUNT← 25C;
uivsloop: dblgoto[.+1, uiVS10, alu#0];
          lu← uiMPSTATUS, call[keyCheck];
          call[uiWAKEOFF];
          uiVSCOUNT← (uiVSCOUNT)-1, goto[uivsloop];
**********************************************************
```

```
*We are in the last scan line of a vertical sync pulse.
*Set up uiVSCOUNT for the next field.
*uiLINK has rv420, fetched during VS4.

uiVS10: uiCRWORD← (uiCRWORD) AND NOT (2C); *ProVS ← 0
        OUTPUT[uiCRWORD, uiCREG];
        uiBUFPTR← (uiBUFPTR) OR (100000C);
        LU← LDF[uiCRWORD, 17, 1];
        T← uiLINESPERFIELD, DBLGOTO[uiEVX, uiODX, ALU=0];
uiEVX: uiVSCOUNT← T, GOTO[uiDCBDONE];
uiODX: uiVSCOUNT← (ZERO)+(T)+1, GOTO[uiDCBDONE];

        end[uitask];
```

```
* MicroD 8.7 (OS 16) of June 8, 1979
* Last edited by Chang, September 12, 1979  10:52 AM
*  edited by Chang, July 6, 1979  6:12 PM
* edited by Johnsson, June 18, 1979  5:24 PM

INSERT[OccupiedDefs];

        TITLE[Mesa1Occupied];

* Locations reserved on page 0

        IMRESERVE[0, 0, 2];       * buffer trap and fault
        IMRESERVE[0, 100, 56];
*       IMRESERVE[0, 156, 10];    * PNIP
        IMRESERVE[0, 166, 5];
*       IMRESERVE[0, 173, 4];     * PNIP
        IMRESERVE[0, 177, 74];
*       IMRESERVE[0, 300, 32];    * LRJ
        IMRESERVE[0, 335, 23];
        IMRESERVE[0, 361, 17];

* Locations reserved on page 1

*       IMRESERVE[1, 100, 142];  * overlayed linkage location
*       IMRESERVE[1, 273, 1];            * overlayed linkage location

* Locations reserved on page 2

        IMRESERVE[2, 0, 100];     * Reserved for RS232
*       IMRESERVE[2, 100, 172];  * overlayable init code
        IMRESERVE[2, 310, 60];    * Reserved for RS232
*       IMRESERVE[2, 372, 1];     * overlayable init code
*       IMRESERVE[2, 374, 1];     * overlayable init code
*       IMRESERVE[2, 376, 1];     * overlayable init code

* Locations reserved on page 3

        IMRESERVE[3, 0, 367];
        IMRESERVE[3, 370, 10];

* Locations reserved on page 7

*       IMRESERVE[7, 27, 1];      * overlayed linkage location
*       IMRESERVE[7, 76, 1];      * overlayed linkage location

* Locations reserved on page 10B

        IMRESERVE[10, 0, 363];

* Locations reserved on page 12B

        IMRESERVE[12,.0, 361];

* Locations reserved on page 14B

        IMRESERVE[14, 0, 137];

* Locations reserved on page 15B

*       IMRESERVE[15, 300, 1];   * overlayed linkage location

* Locations reserved on page 16B

*       IMRESERVE[16, 0, 370];   * initialization

* Locations reserved on page 17B

        IMRESERVE[17, 0, 140];   * kernel
        IMRESERVE[17, 0, 40];            * Key Translation Table

        END;
```

```
* MicroD 8.6 (OS 16) of April 27, 1979
*   at 18-Jun-79 16:08:59
* Last edited by Chang, July 16, 1979  3:16 PM
* edited by Johnsson, June 18, 1979  4:27 PM

INSERT[OccupiedDefs];

        TITLE[Mesa2Occupied];

* Locations reserved on page 0

        IMRESERVE[0, 0, 2];      * buffer trap and fault
        IMRESERVE[0, 2, 12];
*       IMRESERVE[0, 156, 10];  * PNIP
*       IMRESERVE[0, 173, 4];   * PNIP
*       IMRESERVE[0, 300, 32];  * LRJ

* Locations reserved on page 2

*       IMRESERVE[2, 100, 63];

* Locations reserved on page 4

        IMRESERVE[4, 0, 340];
        IMRESERVE[4, 341, 1];
        IMRESERVE[4, 345, 1];
        IMRESERVE[4, 351, 1];
        IMRESERVE[4, 355, 1];
        IMRESERVE[4, 361, 1];
        IMRESERVE[4, 365, 1];
        IMRESERVE[4, 371, 1];
        IMRESERVE[4, 375, 1];
        IMRESERVE[4, 377, 1];

* Locations reserved on page 5

        IMRESERVE[5, 0, 400];

* Locations reserved on page 6

        IMRESERVE[6, 0, 362];
        IMRESERVE[6, 365, 1];
        IMRESERVE[6, 371, 1];
        IMRESERVE[6, 375, 1];
        IMRESERVE[6, 377, 1];

* Locations reserved on page 7

*       IMRESERVE[7, 0, 400];

* Locations reserved on page 11B

*       IMRESERVE[11, 0, 366];

* Locations reserved on page 13B

*       IMRESERVE[13, 0, 400];

* Locations reserved on page 14B

        IMRESERVE[14, 137, 241];

* Locations reserved on page 15B

*       IMRESERVE[15, 0, 374];
*       IMRESERVE[15, 377, 1];

* Locations reserved on page 16B

*       IMRESERVE[16, 0, 357];

* Locations reserved on page 17B

        IMRESERVE[17, 0, 140];

        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        TITLE[mj];

*       MESA JUMP AND ARITHMETIC INSTRUCTIONS
* last modified by Johnsson, June 11, 1979  11:27 AM
* last modified by Sandman, May 9, 1979  2:24 PM
* last modified by Sandman, March 13, 1979  6:30 PM

%
This code assumes that PCB,,PCBh is a base register pair pointing
to the current instruction quadword.  The low two bits of PCB are 0,
and the low 3 bits of the PC (which point to a byte within the quadword)
are kept in the (hardware) register PCF.

Since code segments cannot cross 64K boundaries, and are limited to
32K words in length, the two bytes of PCBh are forced to be equal,
rather than having the least significant byte differ from the msb
by 1 as is the normal case for base registers.

PCF is incremented by the functions NextInst and NextData.  It is
assumed that the register PCX is loaded automatically from PCF at
the start of every bytecode.

Only the low 3 bits of PCF are loaded by PCF←, but PCF and PCX
contain 4 bits so that overflow will be  handled properly.
%

MesaRefill6: gotop[MesaRefill], Pfetch4[PCB,IBUF,4], at[3377]; *refill for page 6

        ONPAGE[0];

*Buffer refill.
MesaRefill:
        PCB ← (PCB) + (4C);
        PCF ← RZero;

*=******* Start of Pilot Code ********************************************
        nop; * hold page fault on page 0
        return;
*=******* End of Pilot Code **********************************************

*//******* Start of Alto Code *******************************************
SwapBytes:                                                              *
        IBuf ← lcy[IBuf, 10];                                           *
        IBuf1 ← lcy[IBuf1, 10];                                         *
        IBuf2 ← lcy[IBuf2, 10];                                         *
        IBuf3 ← lcy[IBuf3, 10], RETURN;                                 *
*//******* End of Alto Code *********************************************
```

```
        ONPAGE[6];
%
Jn, n=2-8.  PCF points to the byte beyond the opcode when execution
starts, so if PCF is odd, the opcode is in the even byte of the
current word, if PCF is even, the opcode is in the odd byte of the
previous word.  The word displacement of the target from PCF[0:2]
and the final lsb of the PC are:

        n                 PCF      PCF
                          even     odd

        2                 0,1      1,0
        3                 1,0      1,1
        4                 1,1      2,0
        5                 2,0      2,1
        6                 2,1      3,0
        7                 3,0      3,1
        8                 3,1      4,0
        9                 4,0      4,1
%
J2:     T←0C, GETRSPEC[127], dblgoto[JnE,JnO,Reven], opcode[200];
J3:     T←1C, goto[Jnx], opcode[201];
J4:     T←1C, GETRSPEC[127], dblgoto[JnE,JnO,Reven], opcode[202];
J5:     T←2C, goto[Jnx], opcode[203];
J6:     T←2C, GETRSPEC[127], dblgoto[JnE,JnO,Reven], opcode[204];
J7:     T←3C, goto[Jnx], opcode[205];
J8:     T←3C, GETRSPEC[127], dblgoto[JnE,JnO,Reven], opcode[206];
J9:     T←4C, goto[Jnx], opcode[207];

JnX:    T← (ldf[GETRSPEC[127],14,3]) + (T), goto[JnOcom];

JnE:    T← (ldf[GETRSPEC[127],14,3]) + (T), goto[JnECom];
JnO:    T← (ldf[GETRSPEC[127],14,3]) + (T) +1, goto[JnECom];

JnECom: Pfetch4[PCB,IBUF];
        T← PCB ← T;       *bypass kludge
        GETRSPEC[127], dblgoto[JnECo, JnECe, Rodd];

JnECe:  PCB ← (lsh[PCB,1]) + 1, goto[JnFin];
JnECo:  PCB ← (lsh[PCB,1]), goto[JnFin];

JnOCom: Pfetch4[PCB,IBUF];
        T← PCB ← T;       *bypass kludge
        GETRSPEC[127], dblgoto[JnOCo, JnOCe, Rodd];

JnOCo:  PCB ← (lsh[PCB,1]) + 1, goto[JnFin];
JnOCe:  PCB ← (lsh[PCB,1]), goto[JnFin];


*=******* Start of Pilot Code *********************************************
JnFin:  PCF ← PCB; *only the low 3 bits of PCF are loaded               *
        PCB ← T;                                                         *
*=******* End of Pilot Code ***********************************************

*//****** Start of Alto Code **********************************************
JnFin:  PCF ← PCB; *only the low 3 bits of PCF are loaded               *
        PCB ← T, LoadPage[0];                                            *
JSwap:  Callp[SwapBytes];                                                *
        lu ← NextInst[IBUF], call[JnRETx];                               *
*//****** End of Alto Code *************************************************

P6Tail:
JnRET:  lu ← NextInst[IBUF];
JnRETx: PCB ← (PCB) and not (3c), NIRET;
```

```
%
Jump Byte:  alpha is a signed displacement from the opcode
This works for -128 < alpha < 122.
This code (and that for JW) uses the register AllOnes as a
temporary, and resets it when done.
%
JB:     T← (NextData[IBUF])-1, opcode[210];
        T ← (PCXREG) + (T);  *PCX = 0..7 here (the NextData may have caused refill, but
*PCX is still correct, since PCX is loaded when the refill code returns)

*T has the signed BYTE displacement relative to PCB
*If jumping backward in same quadword, negative may have become positive;
*mask T to 8 bits

*=******* Start of Pilot Code ******************************************
JBr:    AllOnes ← (rhmask[AllOnes]) AND T, goto[JBq, noH2bit8];        *
*=******* End of Pilot Code ********************************************

*//******* Start of Alto Code ******************************************
JBr:    T ← (rhmask[AllOnes]) AND T;                                    *
        AllOnes ← (Zero) + (T) + 1, goto[JBq, noH2bit8];                *
*//******* End of Alto Code ********************************************

        PCB ← (PCB) - (200C); *If bit 8 = 1, the quantity in T and AllOnes is the byte
*displacement + 400b.  We subtract 400b bytes from PCB.
JBq:    T ← rsh[AllOnes,1];      *form word displacement

*T has positive WORD displacement relative to PCB
JBy:    Pfetch4[PCB,IBUF];
        PCB ← T;
        PCF ← AllOnes;

*=******* Start of Pilot Code ******************************************
        AllOnes ← (Zero)-1, goto[JnRet];                                *
*=******* End of Pilot Code ********************************************

*//******* Start of Alto Code ******************************************
        AllOnes ← (Zero)-1;                                             *
        LoadPage[0], goto[JSwap];                                       *
*//******* End of Alto Code ********************************************

*Jump Word:  alpha,,beta is a 2's complement displacement from the opcode.
*=******* Start of Pilot Code ******************************************
JW:     lu ← CycleControl ← NextData[IBUF], opcode[211]; *get alpha    *
        T ← NextData[IBUF]; *get beta                                   *
*=******* End of Pilot Code ********************************************

*//******* Start of Alto Code ******************************************
JW:     lu ← GetRSpec[127], skip[R even], opcode[211];                  *
        lu ← NextData[IBuf];                                            *
        T ← NextData[IBUF]; *get beta                                   *
        lu ← CycleControl ← NextData[IBUF]; *get alpha                  *
*//******* End of Alto Code ********************************************

        T ← (lhmask[GetRSpec[127]]) OR T;       *CycleControl is in bits 0:7

*T has signed WORD displacement relative to the opcode

*=******* Start of Pilot Code ******************************************
JWx:    T ← (PCFREG) + (T);                                             *
        AllOnes ← T ← (FormMinus4[AllOnes])+(T)+1, dblgoto[JWp,JWn,ALU>=0]; *-4+T+1 = T-3
*=******* End of Pilot Code ********************************************

*//******* Start of Alto Code ******************************************
JWx:    T ← (PCFREG) + (T);                                             *
        AllOnes ← T ← (AllOnes)+(T), dblgoto[JWp,JWn,ALU>=0]; *T-1      *
*//******* End of Alto Code ********************************************

JWn:    T ← 100000C; *negative displacement - put in the bit that the shift is about to lose
        T ← (rsh[AllOnes,1]) or (T), goto[JBy];
JWp:    T ← (rsh[AllOnes,1]), goto[JBy];
```

```
*Jump Equal n, n=2..9:
JEQ2:   T ← (stack&-1), call [JEQEVtest], opcode[212];
        T ← T, goto[JnECom];  * Load T for bypass kludge

JEQ3:   T ← (stack&-1), call [JEQODtest], opcode[213];
        T ← (Form1[AllOnes]) + (T), goto[JnOCom];

JEQ4:   T ← (stack&-1), call [JEQEVtest], opcode[214];
        T ← (Form1[AllOnes]) + (T), goto[JnECom];

JEQ5:   T ← (stack&-1), call [JEQODtest], opcode[215];
        T ← (Form2[AllOnes]) + (T), goto[JnOCom];

JEQ6:   T ← (stack&-1), call [JEQEVtest], opcode[216];
        T ← (Form2[AllOnes]) + (T), goto[JnECom];

JEQ7:   T ← (stack&-1), call [JEQODtest], opcode[217];
        T ← (Form3[AllOnes]) + (T), goto[JnOCom];

JEQ8:   T ← (stack&-1), call [JEQEVtest], opcode[220];
        T ← (Form3[AllOnes]) + (T), goto[JnECom];

JEQ9:   T ← (stack&-1), call [JEQODtest], opcode[221];
        T ← (Form4[AllOnes]) + (T), goto[JnOCom];

JEQEVtest:
        lu ← (stack&-1) - (T);
        dblgoto[JEQjmp,JnRET,alu=0], T ← (ldf[GETRSPEC[127],17,1]);

JEQODtest:
        lu ← (stack&-1) - (T);
        dblgoto[JEQjmp,JnRET,alu=0], T ← 0C;

JEQjmp:
        T ← (ldf[GETRSPEC[127],14,3]) + (T), return;

*Jump Equal Byte

JEQB:   T ← (Stack&-1),call[stkdif],usectask, opcode[222];
JEQBx:  T ← 2c, dblgoto[Ejmp,Onojmp,ALU=0];

Ejmp:   * T contains length of instuction (2)
        T ← (NextData[IBUF]) - T;
Ejmpx:
        T ← (PCFREG) + (T), goto[JBr];

Onojmp:
        lu ← NextData[IBUF];
Onojmpx:
        lu ← NextInst[IBUF],call[JnRETx];

stkdif: lu ← (stack&-1) - (T), return;
```

```
*Jump Not Equal n, n=2..9:
JNE2:   T ← (stack&-1), call [JNEEVtest], opcode[223];
        T ← T, goto[JnECom];  * Load T for bypass kludge

JNE3:   T ← (stack&-1), call [JNEODtest], opcode[224];
        T ← (Form1[AllOnes]) + (T), goto[JnOCom];

JNE4:   T ← (stack&-1), call [JNEEVtest], opcode[225];
        T ← (Form1[AllOnes]) + (T), goto[JnECom];

JNE5:   T ← (stack&-1), call [JNEODtest], opcode[226];
        T ← (Form2[AllOnes]) + (T), goto[JnOCom];

JNE6:   T ← (stack&-1), call [JNEEVtest], opcode[227];
        T ← (Form2[AllOnes]) + (T), goto[JnECom];

JNE7:   T ← (stack&-1), call [JNEODtest], opcode[230];
        T ← (Form3[AllOnes]) + (T), goto[JnOCom];

JNE8:   T ← (stack&-1), call [JNEEVtest], opcode[231];
        T ← (Form3[AllOnes]) + (T), goto[JnECom];

JNE9:   T ← (stack&-1), call [JNEODtest], opcode[232];
        T ← (Form4[AllOnes]) + (T), goto[JnOCom];

JNEEVtest:
        lu ← (stack&-1) - (T);
        dblgoto[JNEjmp,JNEnojmp,alu#0], T ← (ldf[GETRSPEC[127],17,1]);

JNEODtest:
        lu ← (stack&-1) - (T);
        dblgoto[JNEjmp,JNEnojmp,alu#0], T ← 0C;

JNEjmp:
        T ← (ldf[GETRSPEC[127],14,3]) + (T), return;

JNEnojmp:
        lu ← NextInst[IBUF], call[JnRETx];


*Jump Not Equal Byte
JNEB:   T ← (Stack&-1),call[stkdif], usectask, opcode[233];
JNEBx:  T ← 2c, dblgoto[Ojmp,Enojmp,ALU#0];

Ojmp:   * T contains length of instuction (2)
        T ← (NextData[IBUF]) - T, call[Ejmpx];

Enojmp:
        lu ← NextData[IBUF],call[Onojmpx];
```

```
*Jump Less Byte - jump if (TOS-1) < TOS
JLB:    T ← (Stack&-1),call[stkdif],usectask, opcode[234];
JLBx:   T ← (RZero)+1, dblgoto[JLBpos,JLBneg,ALU>=0], FREEZERESULT;      * T ← 1

JLBpos: T ← (RZero) + (T) + 1, dblgoto[Onojmp,Ejmp,NOOVF];                * T ← 2
JLBneg: T ← (RZero) + (T) + 1, dblgoto[Ojmp,Enojmp,NOOVF];                * T ← 2


*Jump Greater Equal Byte
JGEB:   T ← (Stack&-1),call[stkdif],usectask, opcode[235];
JGEBx:  T ← (RZero)+1, dblgoto[JGEBpos,JGEBneg,ALU>=0], FREEZERESULT;   * T ← 1

JGEBpos:        T ← (RZero) + (T) + 1, dblgoto[Enojmp,Ojmp,OVF];                  * T ← 2
JGEBneg:        T ← (RZero) + (T) + 1, dblgoto[Ejmp,Onojmp,OVF];                  * T ← 2


*Jump Greater Byte
JGB:    Stack&-1,usectask,call[stksw], opcode[236];
        lu ← (Stack&-2) - (T), goto[JLBx];


*Jump Less Equal Byte
JLEB:   Stack&-1,usectask,call[stksw], opcode[237];
        lu ← (Stack&-2) - (T), goto[JGEBx];


stksw: T ← stack&+1, return;


*Jump Unsigned Less Byte
JULB:   T ← (Stack&-1), usectask,call[stkdif], opcode[240];
JULBx:  T ← 2c, dblgoto[Onojmp,Ejmp,Carry];

*Jump Unsigned Greater Equal Byte
JUGEB:  T ← (Stack&-1), usectask,call[stkdif], opcode[241];
JUGEBx: T ← 2c, dblgoto[Ojmp,Enojmp,Carry];

*Jump Unsigned Greater Byte
JUGB:   Stack&-1,usectask,call[stksw], opcode[242];
        lu ← (Stack&-2) - (T), goto[JULBx];

*Jump Unsigned Less Equal Byte
JULEB:  Stack&-1,usectask,call[stksw], opcode[243];
        lu ← (Stack&-2) - (T), goto[JUGEBx];

*Jump Zero Equal Byte
JZEQB:  lu ← (Stack&-1), goto[JEQBx], opcode[244];

*Jump Zero Not Equal Byte
JZNEB:  lu ← (Stack&-1), goto[JNEBx], opcode[245];
```

```
*Jump Indexed Byte
*=******* Start of Pilot Code ********************************************
JIB:    T ← Stack&-1, usectask, call[stkdif], opcode[246];             *
        goto[JIBnojmp,carry],Stack&+1;                                 *
        lu ← CycleControl ← NextData[IBUF]; *get alpha                 *
        T ← NextData[IBUF]; *get beta                                  *

        T ← (lhmask[GetRSpec[127]]) OR T; *CycleControl is in bits 0:7 *
        T ← (rsh[Stack,1]) + (T),task;                                 *
        PFETCH1[CODE,RTEMP];                                           *
        Stack&-1, dblgoto[JIBl,JIBr,Reven];                            *
                                                                       *
JIbl:   T ← (ldf[RTEMP,0,10]), goto[JWx];                              *
JIBr:   T ← (rhmask[RTEMP]), goto[JWx];                                *
                                                                       *
JIBnojmp:                                                              *
        lu ← NextData[IBUF], call[JIWnojmpx]; *skip alpha              *
                                                                       *
*=******* End of Pilot Code **********************************************

*//******* Start of Alto Code **********************************************
JIB:    T ← sUnimplemented, goto[doTrapP6], opcode[246];               *
*//******* End of Alto Code ************************************************


*Jump Indexed Word
*=******* Start of Pilot Code ********************************************
JIW:    T ← Stack&-1, usectask, call[stkdif], opcode[247];             *
        goto[JIWnojmp,carry],Stack&+1;                                 *
        lu ← CycleControl ← NextData[IBUF]; *get alpha                 *
        T ← NextData[IBUF]; *get beta                                  *
*=******* End of Pilot Code **********************************************

*//******* Start of Alto Code **********************************************
JIW:    lu ← GetRSpec[127], skip[R even], opcode[247];                 *
        lu ← NextData[IBuf];                                           *
        T ← Stack&-1;                                                  *
        usectask, call[stkdif];                                        *
        goto[JIWnojmp,carry],Stack&+1;                                 *
        T ← NextData[IBUF]; *get beta                                  *
        lu ← CycleControl ← NextData[IBUF]; *get alpha                 *
*//******* End of Alto Code ************************************************

        T ← (lhmask[GetRSpec[127]]) OR T;        *CycleControl is in bits 0:7
        T ← (Stack&-1) + (T),task;
        PFETCH1[CODE,RTEMP];
        T ← RTEMP, goto[JWx];

JIWnojmp:
        lu ← NextData[IBUF]; *skip alpha
JIWnojmpx:
        Stack&-1; *adjust the stack
        lu ← NextData[IBUF], call[JnRET]; *skip beta
```

```
*ADD
@ADD:   T ← Stack&-1, opcode[250];
Addx:   lu ← NextInst[IBuf];
        Stack ← (Stack) + (T), NIRet;

*SUB
@SUB:   T ← Stack&-1, opcode[251];
Subx:   lu ← NextInst[IBuf];
        Stack ← (Stack) - (T), NIRet;

*Multiply - The high half of the 32-bit product is left above the top of the stack
* product low in Stack, hi in RTEMP1
* multipliplicand low in RTEMP, hi in xfMX
* multiplier in xfMY
@MUL:   T ← RTEMP1 ← 0c, opcode[252];
        SALUF ← T, call[PopToT]; * Saluf = 0 is a no op
        xfMY ← T, UseCTask, call[PopToT];
        Stack&+1 ← 0c, skip[alu#0]; * tests xfMY ← T
          Stack&+1 ← 0c, goto[mdPop];
        RTEMP ← T, call[.+1];
        xfMY ← (rsh[xfMY,1]) salufop (T), skip[reven]; * top of loop1
          Stack ← (Stack) + (T), dblgoto[mulDone, MULa, alu=0];
MULb:   T ← RTEMP ← (RTEMP) + (T), FreezeResult, skip[r>=0];
          RTEMP1 ← (RTEMP1) + 1, UseCoutAsCin, goto[mulLong];
        RTEMP1 ← (RTEMP1) + 1, UseCoutAsCin, return;

MULa:   FreezeResult, goto[MULb];

mulLong:
        xfMX ← 1c, call[.+1];
        lu ← (xfMY) salufop (T), skip[reven]; * top of loop2
          Stack ← (Stack) + (T);
        RTEMP1 ← (RTEMP1) + 1, UseCoutAsCin;
        RTEMP ← (RTEMP) + (T);
        T ← xfMX, FreezeResult;
        xfMX ← (xfMX) + (T) + 1, UseCoutAsCin;
        xfMY ← rsh[xfMY,1], skip[reven];
          T ← RTEMP1 ← (RTEMP1) + (T), dblgoto[mdPush, MULc, alu=0];
        T ← RTEMP, return;

MULc:   T ← RTEMP, return;

mulDone: T ← (RTEMP1) + 1, UseCoutAsCin, goto[mdPush];

mdPush: Stack&+1 ← T, goto[mdPop];
mdPop:  Stack&-1, goto[P6Tail];

PopToT: T ← Stack&-1, FreezeResult, return;

*Double
DBL:    T ← lsh[Stack&-1,1], Opcode[253];
PushTP6:        lu ← NextInst[IBUF];
        Stack&+1 ← T, NIRET;
```

```
*Divide - (TOS-1)/TOS. Single word dividend, single word divisor, no check for overflow.
*The remainder is left above the stack.
@DIV:   MNBR ← Stack&-1, opcode[254];
        T ← 0c, goto[LDIVx];

*Long Divide - (TOS-1),,(TOS-2)/TOS. Double word dividend, single word divisor, no check for overflow.
*The remainder is left above the stack.
* dividend low in Stack; hi in RTEMP
* divisor in T
* quotient appears in Stack; remainder in RTEMP
@LDIV:  MNBR ← Stack&-1, call[PopToT], opcode[255];
LDIVx:  RTEMP ← T, LoadPage[opPage0];
        T ← MNBR, gotop[.+1];
        onpage[opPage0];
        lu ← (RTEMP) - (T), goto[zerodivide, alu=0];
        goto[dividecheck, carry];
        nop;
        rcnt ← 17c, call[divStart];
        lu ← RTEMP1; * top of loop
        RTEMP ← (RTEMP) - (T), skip[alu=0]; * subtract divisor
          Stack ← (lsh[Stack,1]) + 1, dblgoto[divs1, divs0, r<0]; * q bit 1
        skip[nocarry];
          Stack ← (lsh[Stack,1]) + 1, dblgoto[divs1, divs0, r<0]; * q bit 1
        RTEMP ← (RTEMP) + (T); * add back divisor
divStart:
        Stack ← lsh[Stack,1], dblgoto[divs1, divs0, r<0]; * q bit 0

divs1:  rcnt ← (rcnt) - 1, goto[divDone1, r<0]; * shift 1
        RTEMP ← (lsh[RTEMP,1]) + 1, dblgoto[divhs1, divhs0, r<0];

divs0:  rcnt ← (rcnt) - 1, goto[divDone2, r<0]; * shift 0
        RTEMP ← lsh[RTEMP,1], dblgoto[divhs1, divhs0, r<0];

divhs1: RTEMP1 ← 1c, return; * next quotient bit known to be 1
divhs0: RTEMP1 ← 0c, return; * next quotient bit unknown

divDone1:       T ← RTEMP, LoadPage[opPage2], goto[divPush];
divDone2:       T ← RTEMP, LoadPage[opPage2], goto[divPush];
divPush:        Stack&+1 ← T, gotop[mdPop];

dividecheck:    T ← sDivideCheck, goto[doTrapP4];
zerodivide:     T ← sZeroDivisor, goto[doTrapP4];
        onpage[opPage2];
```

```
*Negate
@NEG:   T ← Stack&-1, Opcode[256];
        T ← (Zero) - (T), goto[PushTP6];

*Increment
@INC:   T ← (Stack&-1) + 1, goto[PushTP6], Opcode[257];

*And
@AND:   T ← Stack&-1, Opcode[260];
        Stack ← (Stack) and (T), goto[P6Tail];

*OR
@OR:    T ← Stack&-1,.Opcode[261];
        Stack ← (Stack) or (T), goto[P6Tail];

*XOR
@XOR:   T ← Stack&-1, Opcode[262];
        Stack ← (Stack) xor (T), goto[P6Tail];

*Shift
@SHIFT: T ← Stack&-1, Opcode[263];
        dblgoto[ShiftRight,ShiftLeft,ALU<0] , RTEMP ←T ;

ShiftRight:  RTEMP ← (RTEMP) + (17C);
        dblgoto[SHF1,SHF2,Carry];
SHF2:   goto[P6Tail],Stack ← Zero;       *shift count > 17 , use zero
SHF1:   CycleControl ← RTEMP;
        goto[P6Tail],Stack ← RF[Stack];

ShiftLeft:       lu ← (RTEMP) and not (17C);
        dblgoto[SHF3,SHF4,ALU=0], T ← (RTEMP) ;
SHF4:   goto[P6Tail],Stack ← Zero ;      *shift count > 17 , use zero

*T has positive count.  form 0,,-count, then use WFA
SHF3:   RTEMP ← (Zero) - (T) - 1 ;
        RTEMP ← (RTEMP) and (17C) ;
        CycleControl ← RTEMP ;
        goto[P6Tail],Stack ← WFA[Stack];
```

```
*Double Add
@DADD:  MNBR ← Stack&-1, call[GetTDecStk2], opcode[264]; *point to lsb of top doubleword
        Stack ← (Stack) + (T); * add low bits
        Stack&+1, goto[dAddC, carry];
        T ← MNBR, goto[Addx]; * pick up high bits of top doubleword
dAddC:  T ← (MNBR) + 1, goto[Addx]; * pick up high bits of top doubleword

*Double Subtract
@DSUB:  MNBR ← Stack&-1, call[GetTDecStk2], opcode[265]; *point to lsb of top doubleword
        Stack ← (Stack) - (T); * subtract low bits
        Stack&+1, goto[dSubC, NoCarry]; *point to msb of second doubleword
        T ← MNBR, goto[Subx]; *remember msb of top doubleword (TOS)
dSubC:  T ← (MNBR) + 1, goto[Subx];

GetTDecStk2:    T ← (Stack&-2), return; *grab it, point to lsb of second doubleword

*Double Signed Compare:
*If (TOS-2),,(TOS-3) < TOS,,(TOS-1), push -1
*If (TOS-2),,(TOS-3) = TOS,,(TOS-1), push 0
*If (TOS-2),,(TOS-3) > TOS,,(TOS-1), push 1
*Comparisons are signed
DCOMP:  T ← (Stack&-2) + (100000c), Opcode[266];
        Stack ← (Stack) + (100000c), goto[DUCOMPy];


*Double Compare:
*If (TOS-2),,(TOS-3) < TOS,,(TOS-1), push -1
*If (TOS-2),,(TOS-3) = TOS,,(TOS-1), push 0
*If (TOS-2),,(TOS-3) > TOS,,(TOS-1), push 1
*Comparisons are unsigned
DUCOMP: T ← Stack&-2, Opcode[267];
DUCOMPy:
        lu ← (Stack&+1) - (T); *Compare msb's, point at lsb of high doubleword
        goto[DUCompareLowBits, ALU=0], T ← Stack&-2, FREEZERESULT; *grab lsb of top doubleword,
*point at lsb of second doubleword
DUCompx:        dblgoto[DUCompL, DUCompG, NoCarry];

DUCompL:        T ← (RZero) - 1, goto[DUCompEqual];
DUCompG:        T ← (RZero) + 1, goto[DUCompEqual];

DUCompareLowBits:       T ← (Stack) - (T);
        goto[DUCompEqual, ALU=0], FREEZERESULT;
        dblgoto[DUCompL, DUCompG, NoCarry];

DUCompEqual:    Stack ← T, goto[P6Tail];

*ADD01 - on D0, equivalent to ADD
ADD01:  goto[Addx], T ← Stack&-1, Opcode[270];

*Unused opcodes on page 6
        T ← sUnimplemented, goto[doTrapP6], opcode[271];
        T ← sUnimplemented, goto[doTrapP6], opcode[272];
        T ← sUnimplemented, goto[doTrapP6], opcode[273];
        T ← sUnimplemented, goto[doTrapP6], opcode[274];
        T ← sUnimplemented, goto[doTrapP6], opcode[275];
        T ← sUnimplemented, goto[doTrapP6], opcode[276];
        T ← sUnimplemented, goto[doTrapP6], opcode[277];
doTrapP6:
        LoadPage[opPage3];
        gotop[kfcr];


        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        TITLE[ml];

* modified by Johnsson, October 9, 1979  4:44 PM, RFSL stack error
* modified by Sandman, September 18, 1979  10:34 AM, AR 1708 LINKB
* modified by Chang, September 7, 1979  4:24 PM, move InitEnd to page 0
* modified by Johnsson, June 28, 1979  10:11 AM
* modified by Chang, May 27, 1979  2:09 PM, nail down InitEnd
* modified by Sandman, April 5, 1979  2:43 PM

* End of initialization and start of emulators

MC[PilotRunning, 202];
MC[AltoRunning, 214];
MC[AltoMDS, 0];
MC[PilotMDS, 76];

        ONPAGE[TimerPage];
InitEnd:
*//******* Start of Alto Code *********************************************
        NWW ← 100000c, at[InitEndLoc];
        SMAh ← T ← 0c;
        Carry ← T, LoadPage[0];
        T ← (AltoRunning), callp[PNIP];                                    *
        PFetch1[Nova, xfBrkByte, 1]; * use xfBrkByte as temp
        T ← DMA ← 0c;
        lu ← xfBrkByte;
        xfMX ← 4c, goto[MesaBoot, alu = 0];
        PCBh ← T, loadpage[nePage];
        T ← 1c, gotop[JMP];
*//******* End of Alto Code ***********************************************
*=******* Start of Pilot Code ********************************************
        LoadPage[0], at[InitEndLoc];
        T ← (PilotRunning), callp[PNIP];                                   *
*=******* End of Pilot Code **********************************************
MesaBoot:
*=******* Start of Pilot Code ********************************************
        T ← GLOBALhi ← PilotMDS;                                           *
        xfMX ← 1000c;
        xfMX ← (xfMX) or (376c);                                           *
*=******* End of Pilot Code **********************************************
*//******* Start of Alto Code *********************************************
        T ← GLOBALhi ← AltoMDS;                                            *
*//******* End of Alto Code ***********************************************
        T ← GLOBALhi ← (lsh[GLOBALhi,10]) or (T);
        xfWDC ← 1c;
        NWW ← 0c;
        MDS ← 0c;
        MDShi ← T;
        TickCount ← 3c;
        xfXTSReg ← 0c;
        LOCALhi ← T, LoadPage[opPage3];
        MemStat ← (Normal), gotop[MStart];


*Common tail and refill for instructions on page 4
        ONPAGE[4];
MesaRefill4:    gotop[MesaRefill], Pfetch4[PCB,IBUF,4], at[2377];

MesaBBret:
*//******* Start of Alto Code *********************************************
        lu ← GetRSpec[127], skip[R even];                                  *
        lu ← NextData[IBuf];                                               *
        goto[P4Tail];                                                      *
*//******* End of Alto Code ***********************************************

P4Tail: lu ← NextInst[IBUF]; *common tail for page 4 - MUST READ R to interlock
P4Tailx:        NIRET;


*NOP - should not get here unless we need an interrupt
@NOP:   T ← (RZero)+1, goto[NOPint,IntPending], opcode[0];
        goto[P4Tail];
NOPint: RTEMP ← T, loadpage[7]; * RTEMP ← (PC backup if stopping)
        T ← (GetRSpec[103]) xor (377c), callp[MIPendx]; * T ← Stkp xor 377
        T ← (PCFreg) - 1;
        RTEMP ← T;
        PCF ← RTEMP, goto[NOPpcfneg,alu<0];
        goto[P4Tail];   *wait one instruction for PCF← to take
NOPpcfneg:
        PCB ← (PCB) - (4c), goto[P4Tail];

*  Monitor instructions in xPR.mc

P4Ret:  RETURN;
```

```
*Load Local n, n=0-7
LL0:    Pfetch1[LOCAL,Stack,4], goto[P4Tail], opcode[10] ;
LL1:    Pfetch1[LOCAL,Stack,5], goto[P4Tail], opcode[11] ;
LL2:    Pfetch1[LOCAL,Stack,6], goto[P4Tail], opcode[12] ;
LL3:    Pfetch1[LOCAL,Stack,7], goto[P4Tail], opcode[13] ;
LL4:    Pfetch1[LOCAL,Stack,10], goto[P4Tail], opcode[14] ;
LL5:    Pfetch1[LOCAL,Stack,11], goto[P4Tail], opcode[15] ;
LL6:    Pfetch1[LOCAL,Stack,12], goto[P4Tail], opcode[16] ;
LL7:    Pfetch1[LOCAL,Stack,13], goto[P4Tail], opcode[17] ;

*Load Local Byte
LLB:    T ← NextData[IBUF], opcode[20];
        Pfetch1[LOCAL,Stack], goto[P4Tail];

*Load Local Double Byte
LLDB:   T ← NextData[IBUF], opcode[21];
LLDB1:  Pfetch2[LOCAL,Stack], call[FQT];
*If FQT returns, quadOVF has occurred.  Fetch the first word into RTEMP
        CALL[IncT1], Pfetch1[LOCAL,RTEMP];
        Pfetch1[LOCAL,Stack], goto[DoubleReadTail];

*By this time, we know that the first fetch has not faulted.  If the second one faults,
*the fourth instruction after the Pfetch will be aborted.  If we wait to update the first
*word, we don't have to tell the fault handler anything about the state of the memory.
DoubleReadTail: nop;
        Stack&-1; *Adjust the stackpointer to point to the first word of the pair
DoubleReadTailx:        T ← RTEMP;
        Stack ← T; *deposit the first word
        lu ← NextInst[IBUF];
        Stack&+1, NIRET; *final stkp adjust

IncT1:  T ← (Zero) + (T) + 1, return; *increment T by 1

FQT:    goto[.+2, QuadOVF];
        lu ← NextInst[IBUF], call[P4Tailx];
        Stack&-1, return; *adjust the stackpointer modified by the failed fetch to point
*one below the SECOND word to be fetched.

*Store Local n, n=0-7
SL0:    PStore1[LOCAL,Stack,4], goto[P4Tail], Opcode[22];
SL1:    PStore1[LOCAL,Stack,5], goto[P4Tail], Opcode[23];
SL2:    PStore1[LOCAL,Stack,6], goto[P4Tail], Opcode[24];
SL3:    PStore1[LOCAL,Stack,7], goto[P4Tail], Opcode[25];
SL4:    PStore1[LOCAL,Stack,10], goto[P4Tail], Opcode[26];
SL5:    PStore1[LOCAL,Stack,11], goto[P4Tail], Opcode[27];
SL6:    PStore1[LOCAL,Stack,12], goto[P4Tail], Opcode[30];
SL7:    PStore1[LOCAL,Stack,13], goto[P4Tail], Opcode[31];

*Store Local Byte
SLB:    T ← NextData[IBUF], opcode[32];
        PStore1[LOCAL,Stack], goto[P4Tail];
```

```
*Put Local n, n=0-3.  Equivalent to SLn.Push
PL0:    PStore1[LOCAL,Stack,4],goto[PutTail],opcode[33];
PL1:    PStore1[LOCAL,Stack,5],goto[PutTail],opcode[34];
PL2:    PStore1[LOCAL,Stack,6],goto[PutTail],opcode[35];
PL3:    PStore1[LOCAL,Stack,7],goto[PutTail],opcode[36];
* increment stkp by hand so not caught by interlock
PutTail:
        T ← (GetRSpec[103]) xor (377c);
        RTemp ← (Zero) + T + 1;
        Stkp ← RTemp, goto[P4Tail];

*Load Global n, n=0-7
LG0:    Pfetch1[GLOBAL,Stack,3], goto[P4Tail], opcode[37] ;
LG1:    Pfetch1[GLOBAL,Stack,4], goto[P4Tail], opcode[40] ;
LG2:    Pfetch1[GLOBAL,Stack,5], goto[P4Tail], opcode[41] ;
LG3:    Pfetch1[GLOBAL,Stack,6], goto[P4Tail], opcode[42] ;
LG4:    Pfetch1[GLOBAL,Stack,7], goto[P4Tail], opcode[43] ;
LG5:    Pfetch1[GLOBAL,Stack,10], goto[P4Tail], opcode[44] ;
LG6:    Pfetch1[GLOBAL,Stack,11], goto[P4Tail], opcode[45] ;
LG7:    Pfetch1[GLOBAL,Stack,12], goto[P4Tail], opcode[46] ;

*Load Global Byte
LGB:    T ← NextData[IBUF], opcode[47];
        Pfetch1[GLOBAL,Stack], goto[P4Tail];

*Load Global Double Byte
LGDB:   T ← NextData[IBUF], opcode[50] ;
LGDB1:  Pfetch2[GLOBAL,Stack], call[FQT];
*If FQT returns, quadOVF occurred.  See LLDB.
        Call[IncT1], Pfetch1[GLOBAL,RTEMP];
        Pfetch1[GLOBAL,Stack], goto[DoubleReadTail];

*Store Global n, n=0-7
SG0:    PStore1[GLOBAL,Stack,3], goto[P4Tail], Opcode[51];
SG1:    PStore1[GLOBAL,Stack,4], goto[P4Tail], Opcode[52];
SG2:    PStore1[GLOBAL,Stack,5], goto[P4Tail], Opcode[53];
SG3:    PStore1[GLOBAL,Stack,6], goto[P4Tail], Opcode[54];

*Store Global Byte
SGB:    T ← NextData[IBUF], opcode[55];
        PStore1[GLOBAL,Stack], goto[P4Tail];

*Load Immediate n, n=0-6
LI0:    Iu ← NextInst[IBUF], Opcode[56];
        Stack&+1 ← 0c, NIRET;
LI1:    T ← 1c, goto[PushT], Opcode[57];
LI2:    T ← 2c, goto[PushT], Opcode[60];
LI3:    T ← 3c, goto[PushT], Opcode[61];
LI4:    T ← 4c, goto[PushT], Opcode[62];
LI5:    T ← 5c, goto[PushT], Opcode[63];
LI6:    T ← 6c, goto[PushT], Opcode[64];

*Load Immediate Negative 1.
LIN1:   T ← (Zero)-1, goto[PushT], Opcode[65];

*Load Immediate Negative Infinity
LINI:   T ← 100000c, goto[PushT], Opcode[66];

*Load Immediate Byte
LIB:    T ← NextData[IBUF], Opcode[67];
PushT:  Iu ← NextInst[IBUF];
        Stack&+1 ← T, NIRET;

*Load Immediate Word
*=******* Start of Pilot Code ********************************************
LIW:    Iu ← CycleControl ← NextData[IBUF], Opcode[70];                   *
        T ← NextData[IBUF];                                               *
*=******* End of Pilot Code **********************************************

*//****** Start of Alto Code *********************************************
LIW:    Iu ← GetRSpec[127], skip[R even], Opcode[70];                     *
        Iu ← NextData[IBuf];                                              *
        T ← NextData[IBUF];                                               *
        Iu ← CycleControl ← NextData[IBUF];                               *
*//****** End of Alto Code ***********************************************

        T ← (lhmask[GetRSpec[127]]) OR T, goto[PushT];

*Load Immediate Negative Byte
LINB:   T ← 177400c, opcode[71];
        T ← NextData[IBuf] OR T, call[PushT];
```

```
*Local Address Byte - since MDS is 64K aligned, the low half of the base register is L
LADRB:  T ← LOCAL, Opcode[72];
LADRBx: T ← (NextData[IBUF]) + (T), call[PushT];

*Global Address Byte
GADRB:  T ← GLOBAL, goto[LADRBx],  Opcode[73];

*unused opcodes on page 4
        T ← sUnimplemented, goto[doTrapP4], opcode[74];
        T ← sUnimplemented, goto[doTrapP4], opcode[75];
        T ← sUnimplemented, goto[doTrapP4], opcode[76];
        T ← sUnimplemented, goto[doTrapP4], opcode[77];
doTrapP4:
        LoadPage[opPage3];
        gotop[kfcr];
```

```
*Common tail and refill for instructions on page 5
        OnPage[5];
MesaRefill5: gotop[MesaRefill], Pfetch4[PCB,IBUF,4], at[2777];
P5Tail: lu ← NextInst[IBUF];
P5Tailx:        NIRET;


*Read n, n=0-4
R0:     T ← (Stack&-1) + (0c), goto[ReadTail], Opcode[100];
R1:     T ← (Stack&-1) + (1c), goto[ReadTail], Opcode[101];
R2:     T ← (Stack&-1) + (2c), goto[ReadTail], Opcode[102];
R3:     T ← (Stack&-1) + (3c), goto[ReadTail], Opcode[103];
R4:     T ← (Stack&-1) + (4c), goto[ReadTail], Opcode[104];

*Read Byte
@RB:    T ← NextData[IBUF], Opcode[105];
        T ← (Stack&-1) + (T), goto[ReadTail];

ReadTail:       Pfetch1[MDS,Stack],goto[P5Tail];

*Write n, n=0-2
W0:     T ← (Stack&-1) + (0c), goto[WriteTail], Opcode[106];
W1:     T ← (Stack&-1) + (1c), goto[WriteTail], Opcode[107];
W2:     T ← (Stack&-1) + (2c), goto[WriteTail], Opcode[110];

*Write Byte
WB:     T ← NextData[IBUF], Opcode[111];
        T ← (Stack&-1) + (T), goto[WriteTail];

WriteTail:      PStore1[MDS,Stack], goto[P5Tail];

*Read Field
*=******* Start of Pilot Code ********************************************
@RF:    T ← NextData[IBUF], Opcode[112]; *get offset                     *
        lu ← CycleControl ← NextData[IBUF]; *get field descriptor        *
*=******* End of Pilot Code **********************************************

*//****** Start of Alto Code **********************************************
AlignField:                                                              *
        usectask, gotop[AF6];                                            *
        onpage[6];                                                       *
AF6:    T ← apc&apctask;                                                 *
        RTEMP ← T;                                                       *
        lu ← GetRSpec[127], skip[R even];                                *
        lu ← NextData[IBuf];                                             *
        lu ← CycleControl ← NextData[IBUF]; *get field descriptor        *
        T ← NextData[IBUF]; *get offset                                  *
        apc&apctask ← RTEMP;                                             *
        return;                                                          *
                                                                         *
        onpage[5]                                                        *
@RF:    loadpage[6], call[AlignField], Opcode[112];                      *
*//****** End of Alto Code ************************************************

RFy:    T ← (Stack&-1) + (T), call[FetchMDSToStack]; *add pointer, fetch
        lu ← NextInst[IBUF];
RFx:    Stack ← RF[Stack], NIRET; *do the work on the stack

FetchMDSToStack:        Pfetch1[MDS,Stack],RETURN;

*Write Field
*=******* Start of Pilot Code ********************************************
WF:     T ← NextData[IBUF], Opcode[113]; *get offset                     *
        lu ← CycleControl ← NextData[IBUF]; *get field descriptor        *
*=******* End of Pilot Code **********************************************

*//****** Start of Alto Code **********************************************
WF:     loadpage[6], call[AlignField], Opcode[113];                      *
*//****** End of Alto Code ************************************************

Wfz:    T ← (Stack&-1) + (T), call[FetchMDSToRTEMP]; *add pointer, fetch
        RTEMP1 ← T; *save pointer
        T ← WFA[Stack&-1]; *field to be inserted
WFy:    RTEMP ← (WFB[RTEMP]) or (T); *do insert
WFx:    T ← RTEMP1, goto[WS0x];


FetchMDSToRTEMP:        Pfetch1[MDS,RTEMP];
P5Ret:  RETURN; *allow time for T to be written
```

```
*Read Double Byte
RDB:    T ← NextData[IBUF], Opcode[114];
        T ← (Stack&-1) + (T), LoadPage[4], goto[DoubleRead];

*Read Double 0
RD0:    T ← Stack&-1, LoadPage[4], Opcode[115];
DoubleRead: Pfetch2[MDS,Stack], callp[FQT];
*If FQT returns, quadOVF has occurred.  See LLDB.
        Call[IncT1p5], Pfetch1[MDS,RTEMP];
        Pfetch1[MDS,Stack];
DoubleReadx:    LoadPage[4];
        Stack&-1, gotop[DoubleReadTailx];

IncS2T1:        Stack&+2;       *Increment stkp by 2, and...
IncT1p5:        T ← (Zero) + (T) + 1, RETURN; *increment T by 1

*Write Double Byte
WDB:    T ← NextData[IBUF], opcode[116];
        T ← (Stack&-1) + (T);

DoubleWrite:  PStore2[MDS,Stack], call[SQT];
*If SQT returns, do two single stores
        PStore1[MDS,Stack], call[IncS2T1];
        PStore1[MDS,Stack], goto[WSDBx];

*Write Double 0
WD0:    T ← Stack&-1, goto[DoubleWrite], opcode[117];

SQT:    goto[.+2, quadOVF];
        lu ← NextInst[IBUF], call[P5Tailx];
        Stack&+1, return; *set stkp to point to the first word to be stored

*Read String
RSTR:   T ← CNextData[IBUF], call[RWSTRx], opcode[120]; *get alpha
        lu ← RTEMP1, dblgoto[RSTRLeft, RSTRRight, Reven], LoadPage[4];
RSTRLeft:       T ← rsh[RTEMP,10], gotop[PushT];
RSTRRight:      T ← rhmask[RTEMP], gotop[PushT];

*Write String
WSTR:   T ← CNextData[IBUF], call[RWSTRx], opcode[121];
        RTEMP1 ← T, dblgoto[WSTRLeft, WSTRRight, Reven]; *test low bit of RTEMP1, save pointer
WSTRLeft:       T ← lsh[Stack&-1, 10];
        RTEMP ← (rhmask[RTEMP]) or (T), goto[WFx];
WSTRRight:      T ← rhmask[Stack&-1];
        RTEMP ← (lhmask[RTEMP]) or (T), goto[WFx];

RWSTRx: T ← (Stack&-1) + (T); *string index
        RTEMP1 ← T;
        T ← (Stack&-1); *pointer
        T ← (rsh[RTEMP1,1]) + (T), goto[FetchMDSToRTEMP];
```

```
*Read Indexed by Local Pair
RXLP:   T ← NextData[IBUF], opcode[122];
        RTEMP1 ← T, call[LPPointer];
        T ← (Stack&-1) + (T), goto[RILPx];

*Write Indexed by Local Pair
WXLP:   T ← NextData[IBUF], opcode[123];
        RTEMP1 ← T, call[LPPointer];
        T ← (Stack&-1) + (T), goto[WILPx];

*Read Indirect Local Pair
RILP:   T ← NextData[IBUF], opcode[124];
        RTEMP1 ← T, call[LPPointer];
RILPx:  T ← (RTEMP) + (T), goto[ReadTail];

*Read Indirect Global Pair
RIGP:   T ← NextData[IBUF], opcode[125];
        RTEMP1 ← T, call[GPPointer];
RIGPx:  T ← (RTEMP) + (T), goto[ReadTail];

*Write Indirect Local Pair
WILP:   T ← NextData[IBUF], opcode[126];
        RTEMP1 ← T, call[LPPointer];
WILPx:  T ← (RTEMP) + (T), goto[WriteTail];

*Read Indirect Local 0
RIL0:   Pfetch1[LOCAL, RTEMP, 4], call[P5Ret], opcode[127];
        T ← RTEMP, goto[ReadTail];

LPPointer:      T ← 4c; *offset of local 0
        T ← (ldf[RTEMP1, 10, 4]) + (T);
        Pfetch1[LOCAL,RTEMP];
LPPointerx:  T ← ldf[RTEMP1, 14, 4], return;

GPPointer:      T ← 3c; *offset of global 0
        T ← (ldf[RTEMP1, 10, 4]) + (T);
        Pfetch1[GLOBAL,RTEMP], goto[LPPointerx];
```

```
*Write Swapped 0
WS0:    T ← (Stack&-1), call[WSPointer], opcode[130]; *T ← data
WS0x:   PStore1[MDS, RTEMP], goto[P5Tail];

*Write Swapped Byte
WSB:    T ← Stack&-1, call[WSPointer], opcode[131];
WSBx:   T ← (NextData[IBUF]) + (T), call[WS0x];

WSPointer:      RTEMP ← T;
        T ← Stack&-1, return; *T ← pointer

*Write Swapped Field
*=******* Start of Pilot Code *******************************************
WSF:    Stack&-1, opcode[132];                                          *
        T ← NextData[IBUF];                                             *
        lu ← CycleControl ← NextData[IBUF];                             *
*=******* End of Pilot Code *********************************************

*//****** Start of Alto Code *******************************************
WSF:    Stack&-1, loadpage[6], call[AlignField], Opcode[132];           *
*//****** End of Alto Code *********************************************

        T ← (Stack&+1) + (T), call[FetchMDSToRTEMP]; *T ← pointer
        RTEMP1 ← T; *save pointer
        T ← WFA[Stack&-2], goto[WFy];

*Write Swapped Double Byte
WSDB:   T ← NextData[IBUF], opcode[133];
        Stack&-2;
        T ← (Stack&+2) + (T); *T ← pointer + alpha
        Pstore2[MDS,Stack];
        goto[.+2, quadOVF];
WSDBx:  stack&-1, goto[P5Tail]; *back up stkp over pointer
*Do two single stores
        Stack&+1;
        Pstore1[MDS, stack], call[IncS2T1];
        Pstore1[MDS, stack];
        Stack&-2, goto[P5Tail]; *back up stkp over lsb(data) and pointer

*Read Field Code
*=******* Start of Pilot Code *******************************************
RFC:    T ← NextData[IBUF], Opcode[134]; *get offset                    *
        lu ← CycleControl ← NextData[IBUF]; *get field descriptor       *
*=******* End of Pilot Code *********************************************

*//****** Start of Alto Code *******************************************
RFC:    loadpage[6], call[AlignField], Opcode[134];                     *
*//****** End of Alto Code *********************************************

        T ← (Stack&-1) + (T), call[FetchCODEToStack];
        lu ← NextInst[IBUF], call[RFx];

FetchCODEToStack:       PFetch1[CODE,Stack], return;

*Read Field Stack
RFS:    T ← (Stack&-1), call[StackFD], opcode[135];

*//****** Start of Alto Code *******************************************
        lu ← GetRSpec[127], skip[Reven];                                *
        lu ← NextData[IBuf];                                            *
*//****** End of Alto Code *********************************************

        CycleControl ← RTEMP, goto[RFy];

*Write Field Stack
WFS:    T ← Stack&-1, call[StackFD], opcode[136];

*//****** Start of Alto Code *******************************************
        lu ← GetRSpec[127], skip[Reven];                                *
        lu ← NextData[IBuf];                                            *
*//****** End of Alto Code *********************************************

        CycleControl ← RTEMP, goto[WFz];

*Displacement is in left byte, FD is in right byte
StackFD:        RTEMP ← T;
        T ← rsh[RTEMP, 10], return; *get displacement
```

```
*Read Byte Long
RBL:    T ← (Stack&-1) and (377c), call[StackLPx], opcode[137];
        T ← NextData[IBUF], call[RILPLx]; *displacement from pointer

*Write Byte Long
WBL:    T ← (Stack&-1) and (377c), call[StackLPx], opcode[140];
        T ← NextData[IBUF], call[WILPLx]; *displacement

*Read Double Byte Long
RDBL:   T ← (Stack&-1) and (377c), call[StackLPx], opcode[141];
        T ← NextData[IBUF];
DoubleReadLong: Pfetch2[LP,Stack];
        goto[.+2,quadOVF];
        lu ← NextInst[IBUF], call[P5Tailx];
*Do two single fetches, the first to RTEMP
        Pfetch1[LP,RTEMP], call[DecS1IncT1];
        Pfetch1[LP,Stack], goto[DoubleReadx];

DecS1IncT1: Stack&-1, goto[IncT1P5];

*Write Double Byte Long
WDBL:   T ← (Stack&-1) and (377c), call[StackLPx], opcode[142];
        T ← NextData[IBUF];
DoubleWriteLong:  Pstore2[LP, Stack], call[SQT];
*Do two single stores
        Pstore1[LP,Stack], call[IncS2T1]; *straigntforward increment of stkp aborts.  We can speed this up by being more devious.
        Pstore1[LP,Stack], goto[WSDBx];

StackLP:        T ← (Stack&-1) and (377c);
StackLPx:       LPhi ← T;
        T ← LPhi ← (lsh[LPhi,10]) + (T) + 1;
        LPhi ← (fixVA[LPhi]) or (T);
        T ← Stack&-1;
        LP ← T, return;
```

```
*Read Indexed by Local Pair Long
RXLPL:  T ← NextData[IBUF], opcode[143];
        RTEMP ← T, call[LocalLP];
        T ← (Stack&-1) + (T), goto[RILPLx];

*Write Indexed by Local Pair Long
WXLPL:  T ← NextData[IBUF], opcode[144];
        RTEMP ← T, call[LocalLP];
        T ← (Stack&-1) + (T), goto[WILPLx];

*Read Indexed by Global Pair Long
RXGPL:  T ← NextData[IBUF], opcode[145];
        RTEMP ← T, call[GlobalLP];
        T ← (Stack&-1) + (T), goto[RILPLx];

*Write Indexed by Global Pair Long
WXGPL:  T ← NextData[IBUF], opcode[146];
        RTEMP ← T, call[GlobalLP];
        T ← (Stack&-1) + (T), goto[WILPLx];

*Read Indirect Local Pair Long
RILPL:  T ← NextData[IBUF], opcode[147];
        RTEMP ← T, call[LocalLP];
RILPLx: Pfetch1[LP,Stack], goto[P5Tail];

*Write Indirect Local Pair Long
WILPL:  T ← NextData[IBUF], opcode[150];
        RTEMP ← T, call[LocalLP];
WILPLx: Pstore1[LP,Stack], goto[P5Tail];

*Read Indirect Global Pair Long
RIGPL:  T ← NextData[IBUF], opcode[151];
        RTEMP ← T, call[GlobalLP];
        Pfetch1[LP,Stack], goto[P5Tail];

*Write Indirect Global Pair Long
WIGPL:  T ← NextData[IBUF], opcode[152];
        RTEMP ← T, call[GlobalLP];
        Pstore1[LP,Stack], goto[P5Tail];

*Format a long pointer from a local selector,,offset pair
LocalLP:        T ← RTEMP1 ← 5c; *rtemp1 is a flag for decLPTR (below).
        T ← (ldf[RTEMP, 10, 4]) + (T); *note - high half of pointer is fetched first
        Pfetch1[LOCAL,LPhi], goto[decLPTR];
LocalLPy:       Pfetch1[LOCAL,LP]; *low half
LocalLPx:       T ← (LPhi) and (377c);
        T ← LPhi ← (lsh[LPhi,10]) +(T) + 1;
        LPhi ← (fixVA[LPhi]) or (T); *if we decide to trap on references to page 377, do it here
        T ← ldf[RTEMP, 14, 4], return;

*Format a long pointer from a global selector,,offset pair
GlobalLP:       T ← RTEMP1 ← 4c;
        T ← (ldf[RTEMP, 10, 4]) + (T); *note - high half of pointer is fetched first
        Pfetch1[GLOBAL,LPhi], goto[decLPTR];
GlobalLPy:      Pfetch1[GLOBAL,LP], goto[LocalLPx]; *low half

decLPTR:        lu ← ldf[RTEMP1,17,1]; *where to return (T is written here)
        T ← (AllOnes) + (T), dblgoto[LocalLPy,GlobalLPy,ALU#0];
```

```
*Read String Long
RSTRL:  T ← CNextData[IBUF], call[RWSTRLx], opcode[153];
        T ← rsh[RTEMP1, 1], call[FetchLPToRTEMP];
        lu ← RTEMP1, dblgoto[RSTRLeft, RSTRRight, Reven], LoadPage[4];

FetchMNBRLPToRTEMP:     T ← MNBR;
FetchLPToRTEMP: Pfetch1[LP,RTEMP], goto[P5Ret]; * allow time to write T

RWSTRLX:        T ← (Stack&-1) + (T);
        RTEMP1 ← T, goto[StackLP]; *RTEMP1 ← String index

*Write String Long
WSTRL:  T ← CNextData[IBUF], call[RWSTRLx], opcode[154];
        T ← rsh[RTEMP1,1], call[FetchLPToRTEMP];
        RTEMP1 ← T, dblgoto[WSTRLLeft, WSTRLRight, Reven]; *test low bit of RTEMP1
WSTRLLeft:      T ← lsh[Stack&-1, 10];
        RTEMP ← (rhmask[RTEMP]) or (T), goto[WFLx];
WSTRLRight:     T ← rhmask[Stack&-1];
        RTEMP ← (lhmask[RTEMP]) or (T), goto[WFLx];


*Read Field Long
*=******* Start of Pilot Code ********************************************
RFL:    MNBR ← CNextData[IBUF], call[StackLP], opcode[155];              *
        lu ← CycleControl ← CNextData[IBUF], call[FetchMNBRLPToStack];   *
*=******* End of Pilot Code **********************************************

*//****** Start of Alto Code ********************************************
RFL:    loadpage[6], call[AlignField], Opcode[155];                      *
        RTEMP ← T, call[StackLP];                                        *
        T ← RTEMP, call[FetchLPToStack];                                 *
*//****** End of Alto Code ***********************************************
*
RFLx:   lu ← NextInst[IBUF], call[RFx];

FetchMNBRLPToStack:     T ← MNBR;
FetchLPToStack: Pfetch1[LP,Stack], goto[P5Ret]; * allow time to write T

*Write Field Long
*=******* Start of Pilot Code ********************************************
WFL:    MNBR ← CNextData[IBUF], call[StackLP], opcode[156];              *
        lu ← CycleControl ← CNextData[IBUF], call[FetchMNBRLPToRTEMP];   *
*=******* End of Pilot Code **********************************************

*//****** Start of Alto Code ********************************************
WFL:    loadpage[6], call[AlignField], Opcode[156];                      *
        RTEMP ← T, call[StackLP];                                        *
        T ← RTEMP, call[FetchLPToRTEMP];                                 *
*//****** End of Alto Code ***********************************************

        RTEMP1 ← T;
WFLy:   T ← WFA[Stack&-1];
        RTEMP ← (WFB[RTEMP]) or (T);
WFLx:   T ← RTEMP1;
        Pstore1[LP,RTEMP], goto[P5Tail];

*Read Field Stack Long
RFSL:   T ← Stack&-1, call[TtoRTEMP], opcode[157];
        CycleControl ← RTEMP, call[StackLP];
        T ← rsh[RTEMP,10];

*//****** Start of Alto Code ********************************************
        lu ← GetRSpec[127], skip[R even];                                *
        lu ← NextData[IBuf];                                             *
*//****** End of Alto Code ***********************************************

        Pfetch1[LP,Stack], goto[RFLx];

*Write Field Stack Long
WFSL:   T ← Stack&-1, call[TtoRTEMP], opcode[160];
        CycleControl ← RTEMP, call[StackLP];
        T ← rsh[RTEMP,10], call[FetchLPToRTEMP];

*//****** Start of Alto Code ********************************************
        lu ← GetRSpec[127], skip[R even];                                *
        lu ← NextData[IBuf];                                             *
*//****** End of Alto Code ***********************************************

        RTEMP1 ← T, goto[WFLy];

TtoRTEMP:       RTEMP ← T, return;
```

```
*Lengthen Pointer
@LP:    T ← Stack, opcode[161];
        goto[PushTP5, ALU#0], T ← rsh[MDShi,10]; *test for NIL
        T ← 0c;
PushTP5:
        lu ← NextInst[IBuf];
        Stack&+1 ← T, NIRET;

*Store Local Double Byte
SLDB:   T ← NextData[IBUF], opcode[162];
        Pstore2[LOCAL,Stack], call[SQT];
        Pstore1[LOCAL,stack], call[IncS2T1];
        Pstore1[LOCAL,Stack], goto[WSDBx];

*Store Global Double Byte
SGDB:   T ← NextData[IBUF], opcode[163];
        Pstore2[GLOBAL,Stack], call[SQT];
        Pstore1[GLOBAL,stack], call[IncS2T1];
        Pstore1[GLOBAL,Stack], goto[WSDBx];

*Push
PUSH:   Stack&+1, goto[P5Tail], opcode[164];

*Pop
POP:    Stack&-1, goto[P5Tail], opcode[165];

*Exchange
EXCH:   T ← Stack&-1, Opcode[166];
        MNBR ← Stack, Stack ← T, NoRegILockOK;
        T ← MNBR, goto[PushTP5];

*=******* Start of Pilot Code *******************************************
*Link Byte - equivalent to PUSHX; LIB; SUB; SLO;, X is above stkp       *
LINKB:  T ← NextData[IBUF], opcode[167];                                 *
        Stack&+1;                                                        *
        Stack ← (Stack) - (T);                                           *
        Pstore1[LOCAL, Stack, 4], goto[P5Tail];                          *
*=******* End of Pilot Code *********************************************

*//****** Start of Alto Code ********************************************
*Link Byte - equivalent to PUSHX; LIB; SUB; SLO;, X is above stkp       *
LINKB:  T ← NextData[IBUF], opcode[167];                                 *
        Stack&+1, LoadPage[6];                                           *
        Stack ← (Stack) - (T), gotop[LNKBP6];                            *
        OnPage[6];                                                       *
LNKBP6: Pstore1[LOCAL, Stack, 4], goto[P6Tail];                          *
        OnPage[5];                                                       *
*//****** End of Alto Code **********************************************

*Duplicate
DUP:    T ← Stack&-1, opcode[170];
        Stack&+2 ← T, goto[P5Tail];

*=******* Start of Pilot Code *******************************************
*NIL Check                                                               *
NILCK:  T ← Stack&-1, Opcode[171];                                       *
NILCKx: Stack&+1, dblgoto[doTrapP5, P5Tail, ALU=0], T ← sPointerFault;   *
                                                                         *
*NIL Check Long                                                          *
NILCKL: T ← Stack&-1, opcode[172];                                       *
        lu ← (Stack) or (T), goto[NILCKx];                               *
                                                                         *
*Bounds Check - Trap if (TOS-1) >= TOS (unsigned)                        *
BNDCK:  Stack&-1, opcode[173];                                           *
        T ← Stack&+1;                                                    *
        lu ← (Stack&-1) - (T) - 1;                                       *
        dblgoto[doTrapP5, P5Tail, NoCarry], T ← sBoundsFault;            *
*=******* End of Pilot Code *********************************************

*//****** Start of Alto Code ********************************************
NILCK:  goto[P5Tail], Opcode[171];                                       *
NILCKL: goto[P5Tail], opcode[172];                                       *
BNDCK:  Stack&-1, goto[P5Tail], opcode[173];                             *
*//****** End of Alto Code **********************************************

*Unimplemented opcodes on page 5
        T ← sUnimplemented, goto[doTrapP5], opcode[174];
        T ← sUnimplemented, goto[doTrapP5], opcode[175];
        T ← sUnimplemented, goto[doTrapP5], opcode[176];
        T ← sUnimplemented, goto[doTrapP5], opcode[177];

doTrapP5:       LoadPage[opPage3];
        gotop[kfcr];


        end;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
                TITLE[xpr];

* Process Microcode; Last Modified by Sandman on April 6, 1979  9:01 AM

*PSB format
MC[CleanUpOffset, 1];
MC[TimeOffset, 2];
MC[FlagsOffset, 3];
MC[FrameOffset, 4];
MC[Priority, 7];
MC[WaitingOnCV, 10];
MC[AbortPending, 40];
MC[TimeoutAllowed, 100];
MC[EnterFailed, 100000];
MC[SizePSB, 5];

*Monitor Lock format
MC[LockBit, 100000];

*Condition Variable format
MC[WWBit, 100000];

*  Constants
MC[CleanQueue1, 0];
MC[CleanQueue2, 1];
MC[NegInfinity, 100000];
MC[CVBase, 40];
MC[TimerBit, 20];
MC[TimeLocation, 344];

*Dispatch bas address
SET[ProcessDisp, LSHIFT[prPage,10]];

%

PRFlags holds opcode dispatch values and general flags.  Its interpretation is:

  bit 0          0 => Clean Queue 1
                 1 => Clean Queue 2

  bit 10         0 => Requeue not done
                 1 => Requeue done

  bits 16,17     0 => Notify
                 1 => Broadcast
                 2 => Naked Notify
                 3 => not used

  bits 12-15     Opcode Dispatch

All flag constants except NakedNotifyFlags are cycled right 1 in order
to set bit 0 if necessary.

%
*Flag values
MC[MEFlags, 0];                  * Dispatch to 0
SET[MEloc, 0];
MC[MREFlags, 21];                * Dispatch to 2, Clean Queue 2
SET[MREloc, 2];
MC[MXWFlags, 51];                * Dispatch to 5, Clean Queue 2
SET[MXWloc, 5];
MC[MXDFlags, 61];                * Dispatch to 6, Clean Queue 2
SET[MXDloc, 6];
MC[NOTIFYFlags, 100];            * Dispatch to 10, Clean Queue 1, Notify
MC[BCASTFlags, 102];             * Dispatch to 10, Clean Queue 1, Broadcast
SET[WakeHeadloc, 10];
MC[REQFlags, 130];               * Dispatch to 13
SET[REQloc, 13];
MC[NakedNotifyFlags, 2];         * Clean Queue 1, Naked Notify, not cycled
MC[RequeueOccured, 200];         * Requeue occured, not cycled
```

```
        ONPAGE[opPage0];

*ME
@ME:    GOTO[CheckLong1], PRFlags ← (MEFlags), opcode[1];       * Monitor Lock

*MRE
@MRE:   GOTO[CheckLong2], PRFlags ← (MREFlags), opcode[2];      * CV, Lock

*MXW
@MXW:   PRFlags ← (MXWFlags) , opcode[3];                       * Time, CV, Lock
MXWx:   T ← Stack&-1;
        GOTO[CheckLong2], Process ← T;

*MXD
@MXD:   GOTO[CheckLong1], PRFlags ← (MXDFlags), opcode[4];      * Monitor Lock

*NOTIFY
@NOTIFY:
        GOTO[CheckLong1], PRFlags ← (NOTIFYFlags), opcode[5];   * CV

*BCAST
@BCAST: GOTO[CheckLong1], PRFlags ← (BCASTFlags), opcode[6];    * CV

*REQUEUE
@REQUEUE:
        PRFlags ← (REQFlags), goto[MXWx], opcode[7];            * Process, Q1, Q2

CheckLong2:
        T ← (373c); * Stkp value for two long pointer operands (4).
        T ← (nStkp) XOR (T);    * Check for stack size of 4.
        T ← RHMask[MDShi], dblgoto[Short2, Long2, ALU#0];

CheckLong1:
        T ← (375c);     * Stkp value for one long pointer operand (2).
        T ← (nStkp) XOR (T);    * Check for stack size of 2.
        T ← RHMask[MDShi], dblgoto[Short1, Long1, ALU#0];

Short2: Queue2hi ← T, call[FixQueue2];
        T ← RHMask[MDShi], goto[Short1];

Long2:  T ← Stack&-1;
        Queue2hi ← T, call[FixQueue2];
        T ← Stack&-1, goto[Short1];

Long1:  T ← Stack&-1;
Short1: Queue1Hi ← T, goto[FixQueue1];

FixQueue2:
        T ← Queue2hi ← (lsh[Queue2hi,10]) + (T) + 1;
        Queue2hi ← (fixVA[Queue2hi]) or (T);
        T ← Stack&-1;
        Queue2 ← T, return;

FixQueue1:
        T ← Stack&-1, TASK;
        Queue1 ← T;
        lu ← (Queue1hi) OR (T);
        T ← Queue1hi, skip[alu = 0];
          T ← Queue1hi ← (lsh[Queue1hi,10]) + (T) + 1;
        Queue1hi ← (fixVA[Queue1hi]) or (T), goto[ProcessOps];


ProcessOps:
        LoadPage[prPage];
        Dispatch[PRFlags, 11, 4], gotop[ProcessDispatch];

        OnPage[prPage];

ProcessDispatch:
        PRFlags ← RCY[PRFlags, 1], Disp[MEnter];
```

```
*************************************************************************
*       Monitor Enter;
*
*               Input
*       Queue1          Base register pointing at monitor queue
*
*               Temps
*       MQ              process handle
*
*               Constants
*       CurrentPSB      21B, address of CurrentPSB
*       LockBit         100000B, lock bit of Monitor Lock
*
*************************************************************************
MEnter: T ← (CurrentPSB), call[Queue1ToMQ], AT[ProcessDisp, MEloc];
        lu ← MQ, goto[MELocked, R >= 0];
        MQ ← (MQ) AND NOT (LockBit), call[MQToQueue1];
MRENoAbort:
        Stack&+1 ← 1C;   * even location
PrTail: LoadPage[opPage0];
        goto[P4Tail];

MELocked:
        PFetch1[PBase, Process], goto[MREnterFailed];


*************************************************************************
*       Monitor ReEnter;
*
*               Input
*       Queue1          Base register pointing at monitor queue
*       Queue2          Base register pointing at condition queue
*
*               Temps
*       PBase           Base register of PSBs
*       Process         process handle
*       RTemp1          temp
*
*               Constants
*       LockBit         100000B, lock bit of Monitor Lock
*       CleanUpOffset   1, offset of cleanup link in PSB
*       FlagsOffset     3, offset of flags and priority in PSB
*       CurrentPSB      21B, address of CurrentPSB
*       sProcessTrap    17B, offset of sProcessTrap in SD
*
*************************************************************************


MREnter:
        T ← (CurrentPSB), call[Queue1ToMQ], AT[ProcessDisp, MREloc];
        call[PBaseToProcess];
        lu ← MQ, goto[MREnterFailed, R >= 0];          * ignore ww bit
        UseCTask, call[CleanUpQueue];                  * Clean up Queue1
        MQ ← (MQ) AND NOT (LockBit), call[MQToQueue1];
        T ← (Process) + (CleanUpOffset), call[ZeroToPBase];
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp1];
        lu ← (RTemp1) AND (AbortPending);
        goto[MRENoAbort, alu = 0];
        T ← (sProcessTrap);
PRTrap: LoadPage[opPage3];
        gotop[kfcr];

MREnterFailed:
        T ← Queue1;
        Queue2 ← T, TASK;
        T ← Queue1hi;
        Queue2hi ← T;
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp1];
        Queue1hi ← (ReadyQhi), call[ReadyInQueue1];
        RTemp1 ← (RTemp1) OR (EnterFailed), call[RTemp1ToPBase];
        UseCTask, call[RequeueSub];
        T ← PRFlags, goto[ReSchedule];
```

```
*************************************************************************
*       Monitor Exit and Depart;
*
*               Input
*       Queue1          Base register pointing at  monitor queue
*
*************************************************************************
MXDepart:
        UseCTask, call[ExitMon], AT[ProcessDisp, MXDloc];
        T ← PRFlags, goto[ReSchedule];

*************************************************************************
*       Exit Monitor;
*
*               Input
*       Queue1          Base register pointing at  monitor queue
*
*               Temps
*       PBase           Base register of PSBs
*       MQ              process handle
*       Process         process handle
*       EMLink          return link
*
*               Constants
*       LockBit         100000B, lock bit of Monitor Lock
*
*************************************************************************
ExitMon:
        T ← APC&APCTask, call[Queue1ToMQ];
        EMLink ← T;
        T ← MQ ← (MQ) AND NOT (LockBit);
        goto[EMUnlock, alu = 0];
        PFetch1[PBase, Process], call[ReadyInQueue2];
        UseCTask, call[RequeueSub];
        call[Queue1ToMQ];               * Requeue may have changed MQ
        goto[EMUnlock];
EMUnlock:
        MQ ← (MQ) OR (LockBit);
        PStore1[Queue1, MQ, 0], call[PRRet];
        APC&APCTask ← EMLink, goto[PRRet];
```

```
************************************************************************
*       Monitor Exit and Wait;
*
*               Input
*       Queue2          Base register pointing at condition queue
*       Queue1          Base register pointing at monitor queue
*       Process         Timeout value
*
*               Temps
*       PBase           Base register of PSBs
*       MQ              process handle
*       PRTime          holds timeout value
*       RTemp1          process handle
*
*               Constants
*       WWBit           100000B, ww bit of Condition
*       LockBit         100000B, lock bit of Monitor Lock
*       CleanUpOffset   1, offset of cleanup link in PSB
*       TimeOffset      2, offset of flags and priority in PSB
*       FlagsOffset     3, offset of flags and priority in PSB
*       WaitingOnCV     10B, WaitingOnCV bit of PSB
*       AbortPending    40B, AbortPending bit of PSB
*       TimeoutAllowed  100B, TimeoutAllowed bit of PSB
*       CurrentPSB      21B, address of CurrentPSB
*       TimeLocation    344B, address of Timeout time in Memory
*
************************************************************************
MXWait: T ← Process, AT[ProcessDisp, MXWloc];
        PRTime ← T, UseCTask, call[CleanUpQueue];
        T ← QTemp, call[SwapQTempAndQ2];
        UseCTask, call[ExitMon];
        T ← (CurrentPSB), call[PBaseToProcess];
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp1];
        lu ← (RTemp1) AND (AbortPending);
        goto[MXWAbort, alu # 0];
        PFetch1[QTemp, MQ, 0];
        call[PRRet];
        MQ ← (MQ) AND NOT (WWBit), goto[MXWOntoCV, R >= 0];
        PStore1[QTemp, MQ, 0], call[PRRet];
        T ← PRFlags, goto[ReSchedule];
MXWOntoCV:
        lu ← PRTime;
        RTemp1 ← (RTemp1) OR (WaitingOnCV), goto[MXWHaveTime, alu = 0];
        T ← (TimeLocation), call[PBaseToCurrentTime];
        T ← CurrentTime;
        PRTime ← (PRTime) + (T);
        goto[MXWHaveTimex, alu # 0];
        PRTime ← (PRTime) + 1;
MXWHaveTime:
        nop;
MXWHaveTimex:
        Queue1hi ← (ReadyQhi), call[ReadyInQueue1];
        T ← QTemp, call[SwapQTempAndQ2];
        T ← (Process) + (FlagsOffset), call[RTemp1ToPBase];
        UseCTask, call[RequeueSub];
        T ← (Process) + (TimeOffset), call[PRTimeToPBase];
        T ← PRFlags, goto[ReSchedule];
MXWAbort:
        T ← PRFlags, goto[ReSchedule];

SwapQTempAndQ2:
        MNBR ← Queue2, Queue2 ← T, NoRegILockOK;
        T ← MNBR;
        QTemp ← T;
        T ← QTemphi;
        MNBR ← Queue2hi, Queue2hi ← T, NoRegILockOK;
        T ← MNBR;
        QTemphi ← T, return;
```

```
*************************************************************************
*       Notify Broadcast;
*
*               Input
*       Queue1          Base register of queue to be notified
*
*               Temps
*       PBase           Base register of PSBs
*       MQ              process handle
*       RTemp1          process handle
*       Process         process handle
*       PRTime          Did something
*       PRFlags         flags
*
*               Constants
*       WWBit           100000B, ww bit of Condition
*       NegInfinity     100000B
*       FlagsOffset     3, offset of cleanup link in PSB
*       WaitingOnCV     10B, WaitingOnCV bit in PSB
*
*************************************************************************
WakeHead:
        UseCTask, call[CleanUpQueue], AT[ProcessDisp, WakeHeadloc];
WHLoop:
        PRTime ← (Zero), call[Queue1ToMQ];
        MQ ← T ← (MQ) AND NOT (WWBit);
        lu ← LDF[PRFlags, 16, 1], goto[WHExit, alu = 0];
        T ← MQ, call[PBaseToProcess];
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp1];
        PRTime ← (1c), call[ReadyInQueue2];
        RTemp1 ← (RTemp1) AND NOT (WaitingOnCV), call[RTemp1ToPBase];
        UseCTask, call[RequeueSub];
        lu ← PRFlags, goto[WHLoopx, R odd];
        lu ← LDF[PRFlags, 16, 1], goto[WHExit];
WHLoopx:
        goto[WHLoop];
WHExit: T ← PRFlags, goto[ReSchedule, alu = 0];
        lu ← PRTime, dblgoto[SetWWBit, NakedNotified, R even];
SetWWBit:
        MQ ← (WWBit), call[MQToQueue1];          * even location
NakedNotified:
        IntLevel ← (IntLevel) + 1, LoadPage[OpPage0];    * odd location
        goto[CheckCV];
```

```
*************************************************************************
*       Requeue;
*
*       calling instruction must include UseCTask
*
*               Input
*       Queue1          Base register pointing at queue
*       Queue2          Base register pointing at queue
*       Process         process handle
*
*               Output
*       ReSched         BOOLEAN
*
*               Temps
*       PBase           Base register of PSBs
*       Prev            process handle
*       RTemp           Process.link until insert, then Process.priority
*       RTemp1          process handle, priority
*       RTemp2          process handle, priority
*       RLink           return address
*
*               Constants
*       CleanUpOffset   1, offset of cleanup link in PSB
*       FlagsOffset     3, offset of flags and priority in PSB
*       Priority        7, priority bits
*       RequeueOccured  200B
*
*************************************************************************
RequeueOp:
        UseCTask, call[RequeueSub], AT[ProcessDisp, REQloc];
        T ← PRFlags, goto[ReSchedule];

RequeueSub:
        T ← APC&APCTask;
        RLink ← T;
        T ← Process, call[PBaseToRTemp];
        lu ← (RTemp) - (T);
        lu ← Queue1hi, goto[RQGotPP, alu = 0];  * only one process
        goto[RQLoop1, alu = 0];                 * Queue1 NIL, T = Process
        PFetch1[Queue1, Prev, 0], call[PRRet];
        T ← Prev, goto[RQLoop1x];
RQLoop1:
        Prev ← T;
RQLoop1x:
        Prev ← T, call[PBaseToRTemp1];
        T ← RTemp1;
        lu ← (Process) - T;
        goto[RQLoop1x, alu # 0];
        T ← Prev;
        call[RTempToPBase];        * RTemp has Process.link
RQFixCV:
        lu ← Queue1hi;
        T ← (Process) + (CleanUpOffset), goto[RQHaveQ1, alu # 0];
        PStore1[PBase, RTemp];
        call[PRRet];
        goto[RQInsrt];
RQGotPP:
        T ← Prev ← (Zero), goto[RQFixCV];
RQHaveQ1:
        PFetch1[Queue1, RTemp1, 0];
        T ← Process, call[PRRet];
        lu ← (RTemp1) - (T);
        goto[RQInsrt, alu # 0];
        PStore1[Queue1, Prev, 0];
        call[PRRet];
        goto[RQInsrt];
RQInsrt:
        PFetch1[Queue2, Prev, 0], call[PRRet];
        lu ← Prev;
        T ← Process, goto[RQNilPP, alu = 0];
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp];
        T ← (Prev) + (FlagsOffset), call[PBaseToRTemp1];
        RTemp ← (RTemp) AND (Priority);
        T ← (RTemp1) AND (Priority);
        lu ← (RTemp) - T - 1;
        T ← Prev, goto[RQFixQ2, alu < 0];
        goto[RQLoop2];
RQLoop2:
        Prev ← T, call[PBaseToRTemp1];
        T ← (RTemp1) + (FlagsOffset), call[PBaseToRTemp2];
        T ← (RTemp2) AND (Priority);
        lu ← (RTemp) - T - 1;
        T ← RTemp1, dblgoto[RQInsrtHere, RQLoop2, alu >= 0];
RQFixQ2:
        PStore1[Queue2, Process, 0], call[PRRet];
RQInsrtHere:
        T ← Prev;
        call[PBaseToRTemp1];
        T ← Process, call[RTemp1ToPBase];
        T ← Prev, call[ProcessToPBase];
RQRet:  T ← (Process) + (TimeOffset), call[ZeroToPBase];
        APC&APCTask ← RLink;
        PRFlags ← (PRFlags) OR (RequeueOccured), return;
RQNilPP:
        T ← Process;
        call[ProcessToPBase];
        PStore1[Queue2, Process, 0], goto[RQRet];
```

```
**************************************************************************
*       CleanUpQueue;
*         this routine cleans up a queue which may possibly
*         have been left in a mess by Requeue.
*
*       calling instruction must include UseCTask
*
*               Input
*       PRFlags         even => clean Queue1, odd => clean Queue2
*
*               Temps
*       PBase           Base register of PSBs
*       RTemp           process handle
*       RTemp1          process handle
*       MNBR            process handle
*       RLink           return address
*
*               Constants
*       WWBit           100000B, ww bit of Condition
*       CleanUpOffset   1, offset of cleanup link in PSB
*
**************************************************************************
CleanUpQueue:
        T ← APC&APCTask;
        RLink ← T, call[CQueueToRTemp];                 * get pointer to tail
        RTemp ← (RTemp) AND NOT (WWBit);                * ignore ww bit
        T ← (RTemp) + (CleanUpOffset), goto[CURet, alu = 0];
        Pfetch1[PBase,RTemp1], call[PRRet];             * get head of queue
        lu ← T ← RTemp1;
        lu ← (RTemp) - (T), goto[CURetx, alu=0];
        RTemp ← T, goto[CUEmpty, alu = 0];
CULoop: T ← (RTemp) + (CleanUpOffset), call[PBaseToRTemp1];     * get head of queue
        lu ← T ← RTemp1;
        lu ← (RTemp) - (T), goto[CUFoundHead, alu=0];
        goto[CUEmptyx, alu = 0];
        RTemp ← T, goto[CULoop];
CUEmpty:
        RTemp ← (Zero), goto[CUFixCV];
CUEmptyx:
        RTemp ← (Zero), goto[CUFixCV];
CUFoundHead:
        MNBR ← T ← RTemp;
CULoop2:
        RTemp ← T, call[PBaseToRTemp1];
        T ← RTemp1;
        lu ← (MNBR) - (T);
        goto[CULoop2, alu # 0];
        nop;
CUFixCV:
        call[RTempToCQueue];
CURet:  APC&APCTask ← RLink, goto[PRRet];
CURetx: APC&APCTask ← RLink, goto[PRRet];

CQueueToRTemp:
        lu ← PRFlags, DBLGoto[Queue1ToRTemp, Queue2ToRTemp, R >= 0];
RTempToCQueue:
        lu ← PRFlags, DBLGoto[RTempToQueue1, RTempToQueue2, R >= 0];
```

```
*************************************************************************
*       ReSchedule;
*
*               Input
*       T               contains PRFlags
*
*               Temps
*       PBase           Base register of PSBs
*       RTemp           process handle
*       RTemp1          process handle
*       MNBR            process handle
*       RLink           return address
*
*               Constants
*       EnterFailed     100000B, EnterFailed bit of PSB
*       CleanUpOffset   1, offset of cleanup link in PSB
*       FlagsOffset     3, offset of priority and flags in PSB
*       FrameOffset     4, offset of frame in PSB
*       CleanUpOffset   1, offset of cleanup link in PSB
*       Priority        7, priority bits in PSB
*       StkPOffset      10B, offset of stack pointer in state vector
*       DestOffset      11B, offset of dest in state vector
*       CurrentPSB      21B, Current process
*       ReadyQ          22B, Ready queue
*       CurrentState    23B, Current state
*
*************************************************************************


ReSchedule:
        goto[PrTail, NoH2Bit8];
        nop;
IntReSch:
        T ← (CurrentState), call[PBaseToRTemp];
        LoadPage[OpPage3];
        callp[SavPCInFrame];
        T ← RTemp, LoadPage[xfPage1];
        xfTemp ← T, UseCTask, callp[SaveState];
        T ← (CurrentPSB), call[PBaseToRTemp1];   * RTemp1 ← currentPSB
        T ← (RTemp1) + (FrameOffset), call[LocalToPBase];
IdleIntRS:
        T ← (ReadyQ);
        call[PBaseToRTemp1];
        lu ← T ← RTemp1;
        lu ← xfWDC, goto[NoneReady, alu = 0];
        call[PBaseToProcess];                    * Process ← readylist.link

* setup for new process
        T ← (Process) + (FlagsOffset), call[PBaseToRTemp1];
        RTemp ← (xfAV);
        T ← (RTemp) + (FirstStateVector), call[PBaseToRTemp];
        MNBR ← RTemp1, TASK;
        RTemp1 ← T ← (RTemp1) AND (Priority);
        RTemp1 ← (LSH[RTemp1, 2]) + (T);
        T ← (LSH[RTemp1, 1]) + (T), TASK;
        RTemp ← (RTemp) + (T);
        T ← MNBR, goto[NoEnterFailed, R >= 0];
        RTemp1 ← T;
        RTemp1 ← (RTemp1) AND NOT (EnterFailed);
        T ← (Process) + (FlagsOffset), call[RTemp1ToPBase];
        T ← RTemp, call[ZeroToPBase];            * stack[0] ← 0;
        RTemp1 ← 1c;                             * stkp ← 1
        T ← (RTemp) + (StkPOffset), call[RTemp1ToPBase];
NoEnterFailed:
        T ← (Process) + (FrameOffset);
        call[PBaseToRTemp1];
        T ← (RTemp) + (DestOffset), call[RTemp1ToPBase];
        T ← (CurrentPSB), call[ProcessToPBase];          * currentPSB ← process
        T ← (CurrentState), call[RTempToPBase];
        T ← RTemp, LoadPage[xfPage1];
        xfTemp ← T, gotop[LoadState];

NoneReady:
        T ← (sWakeupError), goto[PRTrap, alu # 0];
        RTemp ← (Zero);
        PRFlags ← (Zero), call[IdleLoop];        * clear PRFlags in idle int
IdleLoop:
        goto[IdleInt, IntPending];
PRRet:  return;
IdleInt:
        LoadPage[OpPage3];
        T ← (GetRSpec[103]) XOR (377C), gotop[MIPendx];
```

```
PBaseToProcess:
        PFetch1[PBase, Process], return;
ProcessToPBase:
        PStore1[PBase, Process], return;

PBaseToRTemp:
        PFetch1[PBase, RTemp], return;
RTempToPBase:
        PStore1[PBase, RTemp], return;
PBaseToRTemp1:
        PFetch1[PBase, RTemp1], goto[PRRet];
RTemp1ToPBase:
        PStore1[PBase, RTemp1], goto[PRRet];

PBaseToRTemp2:
        PFetch1[PBase, RTemp2], return;

Queue1ToRTemp:
        PFetch1[Queue1,RTemp,0], return;
RTempToQueue1:
        PStore1[Queue1,RTemp,0], return;

Queue2ToRTemp:
        PFetch1[Queue2,RTemp,0], return;
RTempToQueue2:
        PStore1[Queue2,RTemp,0], return;

Queue1ToMQ:
        PFetch1[Queue1,MQ,0], goto[PRRet];
MQToQueue1:
        PStore1[Queue1,MQ,0], goto[PRRet];

PRTimeToPBase:
        PStore1[PBase, PRTime], return;
ZeroToPBase:
        PStore1[PBase, RZero], return;
LocalToPBase:
        PStore1[PBase, Local], return;

PBaseToCurrentTime:
        PFetch1[PBase, CurrentTime], return;

ReadyInQueue1:
        Queue1 ← (ReadyQ), return;

ReadyInQueue2:
        Queue2hi ← (ReadyQhi);
        Queue2 ← (ReadyQ), return;
```

```
****************************************************************************
*       Interrupt Processing;
*
*               Input
*       WW              Wakeups Waiting register
*       TickCount       number of ticks until scaning PSBs required
*       CurrentTime     current time
*
*               Temps
*       IntLevel        pointer to CV array elements
*       IntType         even => idle loop interrupt
*       ITemp           temp
*       ITemp1          temp
*
*               Constants
*       WWBit           100000B, ww bit of Condition
*       TimeOffset      2, offset of time in PSB
*       FlagsOffset     3, offset of flags and priority in PSB
*       WaitingOnCV     10B, WaitingOnCV bit of PSB
*       NakedNotifyFlags
*
****************************************************************************
* Interrupt Processing

        onpage[OpPage0];

Interrupt:
        WW ← (WW) AND NOT (TimerBit), goto[TimerInterrupt, alu # 0];
        IntLevel ← (CVBase);
CheckCV:
        call[ThisCV];
ThisCV: WW ← RSH[WW, 1], goto[IntThisCV, R odd];
        goto[IntDone, alu = 0]; * load page for done
        IntLevel ← (IntLevel) + 1, return;

IntThisCV:
        T ← IntLevel, Call[IntPBaseToQueue1];
        lu ← Queue1;
        Queue1hi ← (ReadyQhi), goto[DoNothing, alu = 0];
        LoadPage[prPage];
        PRFlags ← (PRFlags) OR (NakedNotifyFlags), goto[WakeHead];

DoNothing:
        IntLevel ← (IntLevel) + 1, goto[CheckCV];

IntDone:
        LoadPage[prPage];       * load page for done
        lu ← IntType, dblgotop[IntReSch, IdleIntRS, R odd];

TimerInterrupt:
        TickCount ← (TickCount) - 1;
        IntLevel ← (CVBase), skip[alu = 0];
          goto[CheckCV];
        ITemp ← (xfAV);
        T ← ITemp ← (ITemp) + (FirstProcess), call[IntPBaseToProcess];
        T ← (ITemp) + 1;
        PFetch1[PBase, ITemp1];
        T ← TimeLocation, call[IntPBaseToCurrentTime];
        CurrentTime ← (CurrentTime) + 1, call[IntCurrentTimeToPBase];
        nop;
TLoop:  T ← (Process) + (TimeOffset);   * can't have call before this
        PFetch1[PBase, ITemp], call[P4Ret];
        lu ← T ← ITemp;
        lu ← (CurrentTime) - (T), goto[CanTimeout, alu # 0];
CheckEnd:
        T ← Process ← (Process) + (SizePSB);
        lu ← (ITemp1) - (T);
        TickCount ← (3c), goto[TLoop, alu >= 0];
        goto[CheckCV];
CanTimeout:
        goto[TimeOut, alu = 0];
        goto[CheckEnd];
Timeout:
        Queue2 ← (ReadyQ);
        Queue1Hi ← (Zero);
        T ← (RZero) + 1, LoadPage[prPage];
        Queue2Hi ← T, UseCTask, call[RequeueSub];
        T ← (Process) + (FlagsOffset), call[IntPBaseToQueue1];
        Queue1 ← (Queue1) AND NOT (WaitingOnCV);
        PStore1[PBase, Queue1], call[P4Ret];
        goto[CheckEnd];

IntPBaseToProcess:
        PFetch1[PBase, Process], goto[P4Ret];
IntPBaseToQueue1:
        PFetch1[PBase, Queue1], goto[P4Ret];

IntPBaseToCurrentTime:
        PFetch1[PBase, CurrentTime], goto[P4Ret];
IntCurrentTimeToPBase:
        PStore1[PBase, CurrentTime], goto[P4Ret];

        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;
insert[globaldefs];
                TITLE[xf];

* modified by Johnsson October 9, 1979  4:08 PM, AR 1829 - PHIP
* modified by Sandman September 18, 1979  2:14 PM, AR 1708 type 2 XFer, PortI
* modified by Chang September 10, 1979  5:17 PM, Bad MapOutofBounds
* modified by Johnsson June 28, 1979  8:10 AM
* modified by Chang June 4, 1979  1:06 PM, fix 3 words, remove MultiDib
* modified by Johnsson May 14, 1979  9:51 AM
* modified by Sandman May 8, 1979  12:37 PM

*Dispatch base addresses
SET[xfPage1p, LSHIFT[xfPage1,10]];
SET[AllocDisp, ADD[xfPage1p, 120]];
SET[xftype, ADD[xfPage1p, 140]];
SET[xfWRtab, ADD[xfPage1p, 160]];
SET[xfRRtab, ADD[xfPage1p, 200]];
SET[MiscDisp, ADD[xfPage1p, 220]];
SET[ReadTimeLoc, ADD[xfPage1p, 0]];
MC[ReadTimeAddr, OR@[170000, ReadTimeLoc]];

*-----------------------------------------------------------------
*       AllocSub
*               called from xferg and ALLOC
*               constants
*       xfav    allocation vector base
*               input registers
*       T       frame size index
*               output registers
*       xfframe, T      frame pointer, xfFrame = 1 if fail
*               temps
*       xffsi   frame size index
*       RTEMP   holds frame chain link
*       xfrsav  holds frame chain link address
*       xfrlink holds return address
*-----------------------------------------------------------------
onpage[xfpage1];
*since xfPage1 has NextData's and NextInsts, it must have refill link at 377
xfRefill:       gotop[MesaRefill], Pfetch4[PCB,IBUF,4], AT[OR@[LSHIFT[xfPage1,10],377]];

SaveRLink:
        xfRLink ← T, return;

AllocSub:
        USECTASK , xffsi ← t;
        T ← APC&APCTASK, call[SaveRLink];       *save link in rlink
        T ← (xfFSI) + (xfav);
ALLOC3:
        task , xfRSAV ← T;
        PFETCH1[MDS,xfframe];   *the head of the list (can't fault)
        DISPATCH[xfframe,16,2];
        DISP[FRTYP0] , xfframe ← T ← (xfframe) AND NOT (3C);
FRTYP0:
        PFETCH1[MDS,RTEMP], AT[ALLOCDISP,0]; * fault if frame swapped out
        T ← xfRSAV, call[xfRet];
        PSTORE1[MDS,RTEMP] ,call[xfret]; * (can't fault)
        APC&APCTASK ← xfRLINK;  *reload link from rlink
        RETURN , T ← xfFrame;
FRTYP1:
        APC&APCTASK ← xfRLINK       , AT[ALLOCDISP,1];
        RETURN , xfFrame ← (1c);
FRTYP2:
        GOTO[ALLOC3A] , T ← xfframe ← RSH[xfframe,2] , AT[ALLOCDISP,2];
FRTYP3:
        GOTO[ALLOC3A] , T ← xfframe ← RSH[xfframe,2] , AT[ALLOCDISP,3];
ALLOC3A:
        goto[ALLOC3] , t ← (xfframe) + (xfav);
```

```
*-----------------------------------------------------------------
*       FreeSub:  Free's the frame if MemStat[14] is 1.
*               constants
*       xfav    allocation vector base
*               input registers
*       T       address of frame
*               output registers
*       xffsi   frame size index
*               temps
*       xfrlink return-to address
*       RTEMP1  holds frame chain link
* Note: None of FreeSub's memory references can page fault!!
*   This implies the av must be resident and L[-1] must not be swapped out.
*-----------------------------------------------------------------

CondFreeSub:    * load lu with FreeFrame bit from MemStat in calling inst.
        goto[FreeSub,alu#0];
        return;
FreeSub:
        xfFrame ← T, USECTASK;
        T ← APC&APCTASK, call[SaveRLink];       *save link in rlink
        T ← (xfFRAME) - 1, task;
        PFETCH1[MDS,xfFSI];             *get fsi
        xfFSI ← T ← (xfFSI) + (xfav), task;
        PFETCH1[MDS,RTEMP1];     * get head of  list from av
        T ← xfFRAME, task;
        PSTORE1[MDS,RTEMP1];     * link ← head
        T ← xfFSI, task;
        PSTORE1[MDS,xfFRAME];           * head ← frame pointer
        APC&APCTASK ← xfRLINK;          *reload link from rlink
        MemStat ← (Normal), RETURN;     * clear FreeFlag
```

```
*-----------------------------------------------------------------
*       Xfer
*                 input registers
*       xfMX    contains dest link or indirect link
*       xftemp2 contains mesa byte program counter
*       xfbrkbyte       contains break op-code to execute
*       xfXTSreg        contains Xfer trap flag
*                 output registers
*       xffsi   holds frame size index
*                 temps
*       xfcount
*       RTEMP
*       xftemp  contains dest link
*-----------------------------------------------------------------
Xfer:
        xftemp←(zero);*setup Xfer trap reason in case of Xfer trap
        T ← xfMX,call[CheckXferTrap];
Xfers:
        xfTemp ← T;
        GOTO[ControlTrap, ALU=0],DISPATCH[xfTemp, 16, 2];
        MemStat ← (MemStat) or (EarlyXfer), DISP[XFERTYPE];
XFERTYPE:
        PFETCH2[MDS,xfTemp], AT[xftype,0];      * Get GFTP & Saved PC.
        call[Loadgc];

*//******* Start of Alto Code *********************************************
        xfTemp ← T ← lsh[xfTemp, 1], goto[pcOkay, r >= 0];
        T ← (Zero) - T;
        xfTemp ← (Zero) + (T) + 1;
*//******* End of Alto Code ***********************************************

pcOkay: T ← xfFrame, goto[XferGo];              * xfFrame contains dest

xfertype1:
        xfCount ← T , AT[xftype,1];     * save descriptor, (xfTemp clobbered)
        xfTemp ← ldf[xfTemp,0,11];      * gfi
        T ← (xfgft);
        task,T ← (lsh[xfTemp,1]) + (T); *t ← gfi*2 + xfgft
        PFETCH2[MDS,xfTemp];
        call[Loadgc];
        T ← (xfTemp);           * evoffset from gft
        xfCount ← (ldf[xfCount,11,5]) + T + 1;  * + evi from descriptor + 1
        xfCount ← lsh[xfCount,1] , goto[xferg]; * 2*evi+2

xfertype2:
        PFETCH2[MDS,RTEMP] , AT[xftype,2];
        T ← xfMX;
        Stack&+1 ← T, TASK;
        Stack&-1;
        T ← RTEMP , goto[Xfers];

xfertype3:
        T ← sUnbound , goto[StashMX] , AT[xftype,3];

ControlTrap:
        GOTO[MTrapF], RTEMP ← sControlFault;
```

```
* Interchange T (the new LOCAL) and LOCAL (the old) and do a
* FreeSub on the old frame if the FreeFlag in MemStat indicates.
* New Byte pc in xfTemp

XferGo:
        MNBR ← LOCAL, LOCAL ← T, task ,NoRegILockOK;
                                        * LOCAL ← new frame, MNBR ← old frame
        T ← MNBR;
        lu ← ldf[MemStat,14,1], call[CondFreeSub]; * load FreeFlag
        PCF ← xfTemp;
        MemStat ← XferFixup; *fixup: byte pc = PC*2+PCF
        t←rsh[xfTemp,1];
        pfetch4[CODE,IBuf];
        PC ← T;                         * bypass kludge
        PC←(PC) and not (3c);

*=******* Start of Pilot Code ******************************************
        t←CODEhi;                                                     *
        PChi←t;                                                       *
*=******* End of Pilot Code ********************************************

*//******* Start of Alto Code ******************************************
        nop;     * allow page fault to happen before load page        *
        t←CODEhi, LoadPage[0];                                        *
        PChi←t, call[SwapBytes];                                      *
*//******* End of Alto Code *********************************************

        LU ← xfBrkByte;
        goto[doBreakByte, ALU#0], MemStat ← (Normal);
* We dispatch the first instruction by hand to insure no interrupt
        T ← NextData[IBuf];
        xfBrkByte ← T, goto[doBreakBytex];

doBreakByte:
        NextData[IBuf];
doBreakBytex:
        T ← xfBrkByte ← (xfBrkByte) OR (40400c);
doBreakBytey:
        xfBrkByte ← LCY[xfBrkByte, 2];
        APC&APCTASK ← xfBrkByte;
        return, xfBrkByte ← ZERO;
```

```
*-----------------------------------------------------------------
*       Loadgc  load global pointer and code pointer given
*               local pointer or GFT pointer
*               constants
*       cpoffset        offset from global frame base to code pointer
*       sCsegSwappedOut trap parameter
*       sUnbound        trap parameter
*               input registers
*       t               mx (local frame pointer or GFT pointer)
*       xfTemp          fetch pending: global frame address
*       xfTemp1         fetch pending: PC or EV offset
*               output registers
*       xfFrame mx (dest) for type0 Xfer, GFT pointer for type1(not used)
*       CODE            code base register
*       GLOBAL  global base register
*       xfTemp          byte program counter or EVoffset*2
*               temps
*       xfRlink         return address
*       xftemp1
*-----------------------------------------------------------------
Loadgc:
        usectask,xfFrame ← T, at[LoadGCLoc];     * xfFrame written into LOCAL after Loadgc Call in type0.
        T ← APC&APCTASK, call[SaveRLink];        *save link in rlink
        t ← (xfTemp);           * new GLOBAL
        GLOBAL ← T;
        pfetch4[MDS, IBuf];     * fetch all of new global overhead
        T ← xfTemp1, task;      * put new PC (or EV offset) in xfTemp
        xfTemp ← T;
        lu ← GLOBAL;
        GOTO [loadgcnull,alu=0] , lu ← LDF[IBuf1,17,1];
        GOTO [loadgcswap,alu#0] , lu ← RSH[IBuf2,10];
        GOTO [ShortCode, ALU#0],  lu ← RSH[IBuf2,6]; * (test) Alto only
        skip [ALU=0], T ← IBuf2;        * test for bad pointer
          T ← IBuf2 ← (IBuf2) OR (100C);
        T ← (LSH[IBuf2,10]) + (T);      * code doesn't cross 64K boundary
LongCode:
        CODEhi ← T, task;
        T ← IBuf1;
        CODE ← T;
        T ← IBuf;
        xfGfiWord ← T;
        apc&apctask←xfRlink,goto[xfRet];
ShortCode:
        T ← MDShi, GOTO [LongCode];
loadgcnull:
        T ← sUnbound , goto[StashMX];
loadgcswap:
        T ← sCsegSwappedOut , goto[StashMX];
```

```
*-----------------------------------------------------------------
*       Xferg  allocates new frame and patches links
*               input registers
*       xfcount index into entry vector
*               output registers
*       T, xfFrame      new frame (has been initialized)
*       CODE    code base register
*       xfTemp  byte program counter
*               temps
*       xfTemp1
*-----------------------------------------------------------------
XferGt:
        t ← xfcount , call[CheckXferTrap];      * might use value from xfTemp
xferg:
        task,t ← xfcount;
        PFETCH2[CODE,xfTemp];   * this can page fault as 1st ref to code!
                                * xfTemp ← wordpc; xfTemp1 ← fsi word
        T ← (ldf[xfTemp1,10,10]), call[AllocSub];       * after this, no more PF's!!
        goto[xfergrf,R odd] ,lu ← xfFRAME;      * new frame is in f
        PSTORE4[MDS,GLOBAL];  * store accesslink and returnlink
        xfTemp ← lsh[xfTemp,1], goto[XferGo];

* Alloc failed, cause trap with destination as parameter
xfergrf:
        t ← xfMX;
        xfATPreg ← t , goto[DoAllocTrap,alu # 0];

* local call, must fabricate procedure descriptor
* xfcount has 2*ev+2, need 4*ev+1
        xfcount ← (lsh[xfcount,1]);     *4*ev+4
        task,xfcount ← (xfcount) - (3c);
        T ← (xfGfiWord) AND NOT (177c);
        t ← (xfcount) + (t);    * '+' is important (don't use OR)
        xfATPreg ← t , goto[DoAllocTrap];
```

```
*-----------------------------------------------------------------------
*       CheckXferTrap  handles Xfer trapping
*               constants
*       sXferTrap       trap parameter
*               input registers
*       T       Xfer trap parameter
*       xfXTSreg        odd implies trap
*       xftemp  Xfer trap reason (0 = other, 200 = LFC, 400 = RET)
*               output registers
*       xfXTPreg        Xfer trap parameter
*       xfXTSreg        Xfer trap reason
*       RTEMP   Xfer trap type
*-----------------------------------------------------------------------
CheckXferTrap:
        goto[noXferTrap,r even] , xfXTSreg←rsh[xfXTSreg,1];
DoXferTrap:
        xfXTPreg ← t;
        task,t ← xftemp;
        xfXTSreg ← t;
        RTEMP ← sXferTrap , goto[MTrapF];

xfRet:
noXferTrap:
        return;
```

```
*-----------------------------------------------------------------
*       SaveState  saves state and zeros stkp
*               called from mstop and DST
*               input registers
*       xfTemp  where to save state
*               temp registers
*       xfRlink return-to address
*       xfTemp1 set stkp to this depth and pop till empty
*       xfTemp2 stkp stored from here
*-----------------------------------------------------------------
SaveState:      * usectask in calling instruction
        T ← APC&APCTASK, call[SaveRLink];       *save link in rlink
        T ← (xfTemp) + (11c);
        Pstore1[MDS, LOCAL], call[sv377];
        T ← (stkp) xor T;       * true stack pointer to T
        xfTemp2 ← T;
        T ← (xfTemp) + (10c), task;
        Pstore1[MDS, xfTemp2];
        T ← (xfTemp2) - (7c);
        xfTemp1 ← 10c, skip[carry];
          xfTemp1 ← (xfTemp1) + T + 1;  * stack depth + 2 (8+(d-7)+1)
        stkp ← xfTemp1, T ← (xfTemp1) - 1;
svloop: call[svpop];
        T ← (stkp) xor T;       * true stack pointer to T
        T ← (AllOnes) + T, goto[svloop, alu#0]; * T ← offset in saved stack
        apc&apctask ← xfRlink, goto[xfRet];

svpop:  T ← (xfTemp) + T;
        Pstore1[MDS, Stack];
sv377:  T ← (377c), return;
```

```
*-----------------------------------------------------------
*       LoadState  loads state
*               input registers
*       xfTemp  where to load state from
*               output registers
*       xfMY    source link
*       xfMX    dest link
*       xfBrkByte       break instruction replacement
*               temp registers
*       xfTemp1 count of stack items to be loaded
*       xfTemp2 count of stack items loaded so far
*-----------------------------------------------------------
LoadState:
        T ← (xfTemp) + (10c);
        Pfetch1[MDS,xfBrkByte], call[xfRot];
        T ← (xfTemp) + (11c);
        Pfetch1[MDS,xfMX];
        NWW ← (NWW) and not (100000c);
        T ← (xfTemp) + (12c);
        Pfetch1[MDS,xfMY], task;
        stkp ← RZero;
        T ← (xfBrkByte) and (17c);
        IBuf ← T;       * stack depth saved for stkp below
        xfTemp1 ← 10c;
        T ← (IBuf) - (7c);      * check for 7 or 8, T ← d-7
        xfBrkByte ← rsh[xfBrkByte,10], skip[carry];
          xfTemp1 ← (xfTemp1) + T + 1;  * stack depth+2 (8+(d-7)+1)
        xfTemp2 ← T ← ZERO, call[ldloop];

ldloop: lu ← (xfTemp1) - T;
        T ← (xfTemp) + T, goto[ldpush, alu#0];
        stkp ← IBuf, goto[Xfer];

ldpush: Pfetch1[MDS, Stack];
        xfTemp2 ← T ← (xfTemp2) + 1, return;  * return to ldloop
```

```
DoAllocTrap:
        RTEMP ← sAllocListEmpty, goto[MTrapF];

* In general, we want to do the FreeSub after all possible Xfer page faults
*  can occur, but before all other traps.

StashMX:                * StackError, UnBound, & cSwappedOut Traps come here
        task,RTEMP ← t;          * RTEMP holds SD index (through FreeSub)
        t ← xfMX;
        xfOTPreg ← t;
MTrapF:                 * XferTrap, AllocTrap, & Control Traps come here
        T ← LOCAL;
        lu ← ldf[MemStat,14,1], call[CondFreeSub]; * load FreeFlag

MTrap:          * PageFault & WriteProtect and RETXfer traps comes to here
        RTEMP ← (RTEMP) + (xfav);
        task,t ← (RTEMP) + (xfsdoffset);
        pfetch1[MDS,xfMX];
        goto[Xfer];


        onpage[opPage3];

MIntTest:       * Back up PC by T if stopping
        goto[MIPend, IntPending];
        return;

MIPend: RTEMP ← T;       * PC backup *interrupt from bitblt enters here
        T ← (GetRSpec[103]) xor (377c); * save Stkp

MIPendx:                * interrupt through NOP enters here
        RTEMP1 ← 342c;   * RSImage
        Stkp ← RTEMP1, RTEMP1 ← T, NoRegILockOK;        * saved Stkp
        T ← Stack ← (Stack) and not (IntPendingBit);
        RS232 ← T;       * turn off IntPending
        Stkp ← RTEMP1;
        lu ← NWW;
        lu ← xfWDC, goto[nothing, alu = 0];
        T ← RTEMP, goto[Disabled, alu#0];        * load PC backup
        IntType ← T, goto[DoInterrupt, alu = 0];
        PRFlags ← (Zero);        * clear flags so that interrupt can or in
*               its state.  In case of no pc backup done in Int Idle loop
        IntType ← (1c), call[P7Ret];
        T ← (PCFreg) - T;
        skip[alu>=0], xfTemp ← T;
         PCB ← (PCB) - (4c);
        PCF ← xfTemp;
DoInterrupt:
        T ← NWW;
        NWW ← (Zero);
        WW ← T, LoadPage[OpPage0];
        lu ← LDF[WW, 13, 1], goto[Interrupt];

Disabled: return;
nothing: return;


*//******* Start of Alto Code *********************************************
*Return to Nova.  IBuf contains the PC at which Nova execution is to begin
Mstop:
        NWW ← (NWW) or (100000c), call[SavPCinFrame];
Mstopx: task,t ← currentstate;
        pfetch1[MDS,xftemp];
        loadpage[xfpage1], T ← RZero;
        usectask,callp[SaveState],PCBH←T;
        RTemp ← FFaultAdd, call[FFS1];
        Stack ← (Stack) and not (1c); * FFAULT ← "crash on page fault"
        DMAh ← T ← 0c;
        SMAh ← T, loadpage[nePage];
        T ← IBuf, gotop[JMP];
*//******* End of Alto Code ***********************************************

*Start from Nova. T has address for LoadState, xfMX even => soft boot xfer
MStart: LOCAL ← 0c, call[FFaultStack]; * LSTFgo loads state from LOCAL+T
        Stack ← (Stack) or (1c);
        lu ← xfMX, goto[LSTFgo, R odd]; * FFAULT ← "trap on page fault"
*//******* Start of Alto Code *********************************************
        Stkp ← RZero;
        loadpage[xfPage1];
        Stack&+1 ← 2c, gotop[xfer];
*//******* End of Alto Code ***********************************************
*=******* Start of Pilot Code *********************************************
        loadpage[xfPage1];
        Stkp ← RZero, gotop[xfer];
*=******* End of Pilot Code ***********************************************

FFaultStack:    RTEMP ← FFaultAdd;
FFS1:   Stkp ← RTEMP, return;
```

```
*-----------------------------------------------------------------
*       SavPCinFrame
*                   input registers
*       PC, PCF             Mesa PC
*                   output registers
*       xfMY                holds Xfer SOURCE
*       LOCAL   holds local base register
*                   temps
*       RTEMP1
*-----------------------------------------------------------------

SavPCinFrame:
        T ← CODE;          *CODEhi and PChi are equal, so we only need to worry about the low word
        T ← (PCB) - (T); *15 bits, since code segments are < 32k.

*=******* Start of Pilot Code *********************************************
        RTEMP1 ← T;
        T ← PCFreg;
        RTEMP1 ← (lsh[RTEMP1,1]) + (T); *byte PC (relative to C) of the instruction to be executed
*when this frame next runs.
*=******* End of Pilot Code ***********************************************

*#******* Start of Alto Code **********************************************
        T ← ldf[GetRSpec[127], 14,3] + T, skip[R even];
        RTEMP1 ← (Zero) - T, goto[StorPC];
        RTEMP1 ← T, goto[StorPC];
*#******* End of Alto Code ************************************************

StorPC: Pstore1[LOCAL, RTEMP1, 1];
        t ← LOCAL;
        xfMY ← t, return;
```

```
*-----------------------------------------------------------------
*       GetLink  fetches control link from global frame or code seg
*               input registers
*       T                       Table offset for link
*       CODE                    code base register
*       GLOBAL  global base register
*               output registers
*       RTEMP                   will contain link
*               temps
*       RTEMP
*-----------------------------------------------------------------
GetLink:
        lu ← xfGfiWord , goto[framelink,R even]; *xfGfiWord was set up by Loadgc when this
*frame became current.
codelink:
        PFETCH1[CODE,RTEMP], return;
framelink:
        PFETCH1[GLOBAL,RTEMP], return;
```

```
        onpage[opPage3];

MesaRefill7:    gotop[MesaRefill], Pfetch4[PCB,IBUF,4], at[3777];


*EFC0 - EFC16
@EFC0:  t ← (1c) , goto[efcr] , OPCODE[300];
@EFC1:  t ← (2c) , goto[efcr] , OPCODE[301];
@EFC2:  t ← (3c) , goto[efcr] , OPCODE[302];
@EFC3:  t ← (4c) , goto[efcr] , OPCODE[303];
@EFC4:  t ← (5c) , goto[efcr] , OPCODE[304];
@EFC5:  t ← (6c) , goto[efcr] , OPCODE[305];
@EFC6:  t ← (7c) , goto[efcr] , OPCODE[306];
@EFC7:  t ← (10c) , goto[efcr] , OPCODE[307];
@EFC8:  t ← (11c) , goto[efcr] , OPCODE[310];
@EFC9:  t ← (12c) , goto[efcr] , OPCODE[311];
@EFC10: t ← (13c) , goto[efcr] , OPCODE[312];
@EFC11: t ← (14c) , goto[efcr] , OPCODE[313];
@EFC12: t ← (15c) , goto[efcr] , OPCODE[314];
@EFC13: t ← (16c) , goto[efcr] , OPCODE[315];
@EFC14: t ← (17c) , goto[efcr] , OPCODE[316];
@EFC15: t ← (20c) , goto[efcr] , OPCODE[317];
@EFCB:  t ← CNextData[IBuf] + 1 , call[efcr], OPCODE[320];

efcr:
        t ← (zero) - t , call[GetLink];
        t ← RTEMP, goto[sfcr];


*LFC1 - LFC16
@LFC1:  GOTO[lfcr] , t ← xfcount ← ( 4C)  , OPCODE[321];
@LFC2:  GOTO[lfcr] , t ← xfcount ← ( 6C)  , OPCODE[322];
@LFC3:  GOTO[lfcr] , t ← xfcount ← (10C)  , OPCODE[323];
@LFC4:  GOTO[lfcr] , t ← xfcount ← (12C)  , OPCODE[324];
@LFC5:  GOTO[lfcr] , t ← xfcount ← (14C)  , OPCODE[325];
@LFC6:  GOTO[lfcr] , t ← xfcount ← (16C)  , OPCODE[326];
@LFC7:  GOTO[lfcr] , t ← xfcount ← (20C)  , OPCODE[327];
@LFC8:  GOTO[lfcr] , t ← xfcount ← (22C) , OPCODE[330];
@LFC9:  GOTO[lfcr] , t ← xfcount ← (24C) , OPCODE[331];
@LFC10: GOTO[lfcr] , t ← xfcount ← (26C) , OPCODE[332];
@LFC11: GOTO[lfcr] , t ← xfcount ← (30C) , OPCODE[333];
@LFC12: GOTO[lfcr] , t ← xfcount ← (32C) , OPCODE[334];
@LFC13: GOTO[lfcr] , t ← xfcount ← (34C) , OPCODE[335];
@LFC14: GOTO[lfcr] , t ← xfcount ← (36C) , OPCODE[336];
@LFC15: GOTO[lfcr] , t ← xfcount ← (40C) , OPCODE[337];
@LFC16: GOTO[lfcr] , t ← xfcount ← (42C) , OPCODE[340];

@LFCB:  T ← NextData[IBuf] , OPCODE[341];
        xfcount ← (zero) + (t) + 1;
        xfcount ← lsh[xfcount,1] , goto[lfcr];

lfcr:
        xfMX ← zero , call[SavPCinFrame];
        loadpage[xfpage1];
        xftemp←(400c),goto[XferGt];*setup xftemp with 2 * Xfer trap reason
```

```
*SFC
@SFC:   goto[sfcr],T ← stack&-1 , OPCODE[342];
sfcr:
        xfMX ← t , call[SavPCinFrame];
        loadpage[xfpage1];
gotoxfer:
        goto[Xfer];

*RET
@RET:   T←(LOCAL)+(6c) , OPCODE[343];*code for XTSreg if Xfer trap
        LoadPage[xfpage1];
        xftemp←(1000c), callp[CheckXferTrap];*setup xftemp with 2 * trap reason in case of Xfer trap
        task,pfetch1[LOCAL, xfMX, 2];    * Fetch destination link (at xfretlinkoffset)
        MemStat ← (FreeFrame);   * set FreeFlag .
        LoadPage[xfpage1], T ← xfMX;
        GOTOp[Xfers], xfMY ← (0C);

*LLKB
@LLKB:  T ← NextData[IBuf] , OPCODE[344];
        call[GetLink] , T ← (zero) - (T) - 1;
        T ← RTEMP, goto[PushTP7];

*PORTO
@PORTO: call[SavPCinFrame] , OPCODE[345];
        T ← stack&-1;
        task, xftemp ← T;
        pstore1[MDS,xfMY];
        task,T ← (zero) + (T) + 1;
        pfetch1[MDS,xfMX];
        T ← xftemp , loadpage[xfpage1];
        goto[Xfer] , xfMY ← T;

*PORTI
@PORTI: Stack&+1, OPCODE[346];
        lu ← xfMY;
        goto[portinz,alu=0] , T ← Stack&-1;
        pstore1[MDS,Rzero],call[P7Ret];
        T ← (RZero) + (T) + 1;
portinz:
        goto[P7Tail],PSTORE1[MDS,xfMY];

*KFCB
@KFCB:
        T ← NextData[IBuf] , OPCODE[347];
kfcr:
        RTEMP ← T, call[SavPCinFrame], at[KFCRLoc];
        loadpage[xfpage1];
        goto[MTrap];

*DESCB
@DESCB:
        T ← xfGfiWord, OPCODE[350];
        RTEMP ← T;
descbcom:
        T ← NextData[IBuf];
        RTEMP1 ← t;
        t ← (RTEMP) and not (177c);
        t ← (lsh[RTEMP1,1]) + (t) + 1;
        goto[P7Tail],stack&+1 ← t;

*DESCBS
@DESCBS:
        T ← (STACK&-1) + (xfgfioffset), OPCODE[351];
        pfetch1[MDS,RTEMP], goto[descbcom];
```

```
*BLT
@BLT:   LP ← 0c, OPCODE[352];
        t←MDShi;
BLTcom:
        LPhi←t;
        * fixup: fetch => count + 1; store => source-1, dest-1, count+1
        MemStat ← BltFixup,call[BLTx]; * set return address to BLTloop;

BLTloop:
        lu ← stack&-1;
        goto[BLTdone,alu=0] , T ← stack&+1; *get source, point to count
        stack ← (stack) - 1; *decrement count
        pfetch1[LP,RTEMP];
        stack&+1; *point to dest
        T ← stack&-2; *get dest, point to source
        pstore1[MDS,RTEMP];
        stack ← (stack) + 1;*increment source
        Stack&+2;
        Stack ← (Stack) + 1, goto[BLTint, IntPending]; *increment dest
BLTx:   stack&-1, return; *point to count, return to BLTloop

BLTdone:        stack&-2;
BLTdonex:       goto[P7Tail],MemStat ← Normal;

BLTint: loadpage[opPage0];
BLTstop:        T ← 1c, gotop[NOPint];

*BLTL
@BLTL:  T ← (Stack&-1) and (377c), OPCODE[353];
        LPdesthi ← T;
        T ← LPdesthi ← (lsh[LPdesthi,10]) + (T) + 1;
        LPdesthi ← (fixVA[LPdesthi]) OR T;
        T ← Stack&-2, loadpage[5];
        LPdest ← T, callp[StackLP];
        RTEMP1 ← (Zero);
        MemStat ← BltLFixup; * fixup: source+T, dest+T, count+1
        Stack&+3, call[BLTLloop];               * point to count

BLTLloop:
        lu ← Stack;             * read count, point to source lo
        T ← RTEMP1, goto[BLTLdone,alu=0];
        pfetch1[LP,RTEMP];
        Stack ← (Stack) - 1;    * decrement count
        RTEMP1 ← (RTEMP1) + 1;  * increment offset
        pstore1[LPdest,RTEMP];
        goto[BLTLint, IntPending];
        return; * goes to BLTLloop

BLTLdone:
        stack&-3, goto[BLTdonex];

BLTLint:
        Stack&-2;
        call[BLTLbump]; * wait for page fault before updating stack
        Stack&+2,call[BLTLbump];
        loadpage[opPage0], goto[BLTstop];

BLTLbump:
        Stack ← (Stack) + (T) + 1;
        Stack&+1, skip[nocarry];
         Stack ← (Stack) + 1, return;
        return;

*BLTC
@BLTC:  T ← CODE, OPCODE[354];
        LP ← T;
        T ← CODEhi, GOTO[BLTcom];

*BLTCL
@BLTCL: T ← sUnimplemented, goto[kfcr], Opcode[355];
```

```
*ALLOC
@ALLOC:
        T ← STACK&-1 , OPCODE[356];
        loadpage[xfpage1];
        callp[AllocSub] , xfATPreg ← t;
        goto[allocrf,R odd] ,lu ← xfFRAME;
PushTP7:        goto[P7Tail], stack&+1 ← t;
P7Ret:  return;

allocrf:
        call[SavPCinFrame];
        loadpage[xfpage1];
        xfATPreg ← lsh[xfATPreg,2] , goto[DoAllocTrap];

*FREE
@FREE:
        T ← Stack&-1,loadpage[xfpage1], OPCODE[357];
        callp[FreeSub], MemStat ← (FreeFrame);

P7Tail: lu ← NextInst[IBuf], at[P7TailLoc];
P7Tailx:        NIRET;

*Increment Wakeup Disable Counter (Disable Interrupts)
@IWDC:  goto[P7Tail],xfwdc ← (xfwdc) + 1 , OPCODE[360];

*Decrement Wakeup Disable Counter (Enable Interrupts)
@DWDC:
        task,t ← (R400) or (52c), OPCODE[361];
        pfetch1[Nova,xftemp];
        xfwdc ← (xfwdc) - 1;
        t ← xftemp;
        NWW ← (NWW) or (t);
        goto[DWDCnone, alu=0];  * see if any interrupts
        RTEMP ← 342c;    * points to RSImage
        T ← (GetRSpec[103]) xor (377c); * read Stkp
        Stkp ← RTEMP, RTEMP ← T;
        T ← Stack ← (Stack) or (IntPendingBit); * set IntPending
        RS232 ← T;
        Stkp ← RTEMP;
* dispatch the next instruction by hand to allow one
* instruction without interrupt
DWDCnone:
        T ← NextData[IBuf];
        xfBrkByte ← T, loadpage[xfPage1];
        T ← xfBrkByte ← (xfBrkByte) OR (40400c),gotop[doBreakBytey];

*STOP
*//******* Start of Alto Code *********************************************
@STOP:  IBuf ← StopStopPC, goto[Mstop] , OPCODE[362];                    *
*//******* End of Alto Code ***********************************************
*=******* Start of Pilot Code ********************************************
@STOP:  T ← sUnimplemented, goto[kfcr] , OPCODE[362];                    *
*=******* End of Pilot Code **********************************************

*CATCH
CATCH:  lu ← NextData[IBUF],call[P7Tail], Opcode[363];
```

```
*MISC - extended opcodes accessed by dispatching on alpha
@MISC:  T ← NextData[IBUF], opcode[364];
        LoadPage[xfPage1], RTEMP ← T;
        gotop[.+1], Dispatch[RTEMP, 14, 4];

        OnPage[xfPage1];
        Disp[@ASSOC];            * dispatch on second byte

MiscTail:       lu ← NextInst[IBUF], AT[MiscDisp,20];
MiscTailx:      NIRET;

* Associate - TOS contains map entry, (TOS-1) contains VP which is to get it.
@ASSOC: T ← (Stack&-1), AT[MiscDisp,0];
        Call[MapLP], xBuf ← T;
ASSOC1: XMap[LP, xBuf, 0], goto[MiscTail];

* Set Flags
@SETF:  T ← (Stack&-1), AT[MiscDisp,1];
        Call[MapLP], xBuf ← T;
        XMap[LP, xBuf, 0];
        T ← LSH[xBuf3,10];                  * Put flags,,card,blk0 in left byte
        T ← xBuf1 ← (RHMASK[xBuf1]) OR (T); * blk1,rowaddr in low byte
        T ← (ZERO) or not (T);              * push old flags & page
        Stack&+1 ← T;                       * push old flags & page
        lu ← (ldf[xBuf3,11,3]) -1;          * =0 if map entry = VACANT
        goto[.+2,ALU#0], xBuf ← (xBuf) and (70000C);    * isolate new flags, ignore LogSE
        goto[ASSOC1], xBuf ← T; * Vacant entry, use old flags, oldpage
        T ← (Stack) and not (170000C);  * Get old page number
        goto[ASSOC1], xBuf ← (xBuf) or (T);     * new flags, old page

*Subroutine MapLP creates a base register pair from a virtual page number for the Map opcodes.
MapLP:  T ← LSH[Stack,10];
        LP ← T;         * Set low Base
        T ← LHMask[Stack&-1];
*       goto[.+2,R>=0], LPhi ← T;       * Set high byte of high base
        goto[.+2,Alu>=0], LPhi ← T;     * Set high byte of high base
        LPhi ← (LPhi) OR (40000C);      * set bit 1 if 0 is set.
MiscRet:
        RETURN;                  * one instruction to allow LPhi to be written



* Read & Write Ram format: stack=40:43,,addr, (stack-1)=40:43, (stack-2)=0:17, (stack-3)=20:37.
@ReadRam:       T ← LDF[Stack&-1, 4, 14], AT[MiscDisp,2];       * get address
        RTEMP ← T;
        call[CSRead], T ← 1C;   * read 20:37
        call[CSRead], T ← 0C;   * read 0:17
        call[CSRead], T ← 3C;   * read 40:43
        T ← RTEMP;
        GOTO[MiscTail], Stack ← (LSH[Stack, 14]) or (T);

* Subroutine CSRead reads control store for ReadRam opcode.
CSRead: APCTASK&APC ← RTEMP;
        ReadCS;
        T ← CSData, AT[MiscDisp,22]; *successor of CSOp must be even
        return, Stack&+1 ← T;

%*************************************************************
@WriteRam:      T ← Stack&-1, AT[MiscDisp,3];    * get 40:43,,address
        RTEMP ← T;
        T ← LDF[RTEMP, 0, 4];   * get 40:43
        LU ← Stack&-1;          * bits 0:17
        APCTASK&APC ← RTEMP;    * value of apctask a don't care
        WriteCS0&2;
        LU ← Stack&-1, AT[MiscDisp,24];         * bits 20:37
        APCTASK&APC ← RTEMP;    * value of apctask a don't care
        WriteCS1, goto[MiscTail];


@JumpRam:       NOP, AT[MiscDisp,4];            * Filler
        APCTASK&APC ← Stack&-1, Call[MiscRet];
* The notified microcode must not task until it is ready to return to emulator!
        CALL[MiscTailx], lu ← NextInst[IBUF];
%*************************************************************
SET[LRJBase, ADD[LSHIFT[LRJpage,10], 300]];
MC[VersionID,0];

MiscTrap:
        LoadPage[OpPage3];
        gotop[kfcr];

@LoadRamJ:
        T ← (stack&-1) xor (1c), AT[MiscDisp,3];
        RTEMP1←T, loadpage[opPage1];    * save bits, jump complemented
        T ← (Stack&-1) and (377c), callp[StackLPx];
        pfetch1[LP,RTEMP,0], call[MiscRet];
        LU ← (RTEMP) xor (VersionID);
        T ← sUnimplemented, GOTO[MiscTrap,alu#0];
        T ← (GetRSpec[103]) xor (377c);
        RTEMP ← FFaultAdd;
        Stkp ← RTEMP, RTEMP ← T;
        Stack ← (Stack) and not (1c);
        loadpage[LRJpage];
        Stkp ← RTEMP, gotop[.+1];

                OnPage[LRJpage];

LRJenter:
        t ← xfTemp ← 1c, AT[LRJBase, 0];
        nop, AT[LRJBase, 1];    * wait for write of xfTemp to avoid bypass problem
```

```
LRJloop:
	pfetch1[LP,xBuf2],call[LRJIncCount], AT[LRJBase, 2];
	pfetch1[LP,xBuf],call[LRJIncCount], AT[LRJBase, 3];
	pfetch1[LP,xBuf1],call[LRJIncCount], AT[LRJBase, 4];
	T ← ldf[xBuf2,0,14], AT[LRJBase, 5];	* address
	xBuf3 ← T, AT[LRJBase, 6];
	lu ← (xBuf3) XNOR (170000c), AT[LRJBase, 7]; *look for m-i address = 7777
	T ← xBuf2, goto[RamLoaded,alu=0], AT[LRJBase, 10];
	LU ← xBuf, AT[LRJBase, 13];
	APC&APCTASK ← xBuf3, AT[LRJBase, 11];
	WRITECS0&2, AT[LRJBase, 14];
	LU ← xBuf1, AT[LRJBase, 15];
	APC&APCTASK ← xBuf3, AT[LRJBase, 16];
	WRITECS1, AT[LRJBase, 17];
	T ← xfTemp, goto[LRJloop], AT[LRJBase, 20];

LRJIncCount:
	T ← xfTemp ← (xfTemp) + 1, goto[JumpRet], AT[LRJBase, 21];

RamLoaded:
	RTEMP1, GOTO[.+2,Rodd], AT[LRJBase, 12];	* odd if no jump
	APCTASK&APC ← (xBuf1), call[JumpRet], AT[LRJBase, 22];	* set TPC for return
	T ← (GetRSpec[103]) xor (377c), AT[LRJBase, 23];
	RTEMP ← FFaultAdd, AT[LRJBase, 24];
	Stkp ← RTEMP, RTEMP ← T, AT[LRJBase, 25];
	Stack ← (Stack) or (1c), AT[LRJBase, 26];
	loadpage[4], AT[LRJBase, 27];
	Stkp ← RTEMP, gotop[P4Tail], AT[LRJBase, 30];

JumpRet:
	RETURN, AT[LRJBase, 31];

		OnPage[xfPage1];
* The following is for byte code CLRDev --- clear all devices and timers

SET[QDBase,lshift[ClrDvPage,10]];

SET[Qdloc,ADD[QDBase,303]];
MC[QdxL,AND@[Qdloc,377]];
MC[QdxH,OR@[150000,AND@[Qdloc,7400]]];

SET[QdretLoc,Add[QDBase,307]];
MC[QdRetL,AND@[QdretLoc,377]];
MC[QdRetH,AND@[QdretLoc,7400]];

@CLRDev:
	RTEMP ← (100000C), AT[MiscDisp,4];
	LoadPage[ClrDvPage];
	RTEMP ← (RTEMP) OR (16C), gotop[ClearTimers]; * clear out all but memory refresh

		OnPage[ClrDvPage];
ClearTimers:
	LOADTIMER[RTEMP];	*Clear out all Timers except one
	NOP;
	NOP;
	NOP;
	task;
	RESETMEMERRS;	*Clear any pending memory errors
	LU ← (RTEMP) AND (17C); *there are 15d timers to be cleared
	RTEMP ← (RTEMP) - 1,GOTO[ClearTimers, ALU#0];

Qdtask:	RTEMP ← QdxL; *Quiesce tasks 15b to 1
	RTEMP ← (RTEMP) or (QdxH);
	Stack&+1 ← QdRetL;
	Stack ← (Stack) or (QdRetH);
Qdloop:	APC&APCTASK ← RTEMP;
CLDret:	return; *goes to Qdx

Qdx:	APC&APCTASK ← stack,call[CLDret], AT[Qdloc]; *Notify comes here. Leave task's TPC pointing at Qdxy.
Qdxy:	goto[CLDret]; * gets here if wakeup

Qdret:	lu ← ldf[RTEMP,0,3], AT[QdretLoc]; *DevIndex points to this location
	RTEMP ← (RTEMP) - (10000C), dblgoto[ZapDevices,Qdloop,ALU=0];

ZapDevices: T ← 177400C;
	RTEMP ← 0c;
ZapDloop:	OUTPUT[RTEMP];	*send a 0 to all registers of all devices, hopefully quiescing them
	T ← (zero) + (T) + 1;
	goto[ZapDloop,ALU<0];
	Stack&-1, loadpage[4];
	gotop[P4Tail];

* Opcodes for Mesa Input/Ouput.  Stack[0:7]=XXX, Stack[10:13]=task no.,
*	Stack[14:17]=I/O register number.

		OnPage[xfPage1];

* Opcodes for Mesa Input/Ouput.  Stack[0:7]=XXX, Stack[10:13]=task no.,
*	Stack[14:17]=I/O register number.

@INPUT: T ← Stack&-1, AT[MiscDisp,5];
	goto[MiscTail],Input[Stack];

@OUTPUT:	T ← Stack&-1, AT[MiscDisp,6];
	goto[MiscTail], Output[Stack];

%
*CheckSum opcode.  Stack, Stack&-1=Long pointer to end of buffer,
*	Stack&-2=negative count, Stack&-3=checksum.

@ChkSumL:	t←stack&-1, loadpage[lpPage2], AT[MiscDisp, 7];
```

```
        lpHighB←t, call[StackLP];        * convert into base reg pair
        stack←(stack)+1, loadpage[lpPage2];      * fudge the count
        lpLowB←(lpLowB)+1;

                onpage[lpPage2];

        goto[.+2, carry];
        goto[ChkSNext], lpHighB←(lpHighB)-(400C)-1;* count goes backwards
        goto[ChkSNext1], t←(stack&-1)-1;
ChkSNext:       t←(stack&-1)-1;                 * count
ChkSNext1:      goto[ChkSEnd, alu=0];
*       task, pfetch1[lpLowB, xftemp];  * get data
        pfetch1[lpLowB, xftemp];        * get data
        t←stack;                        * get current checksum
        xftemp←(xftemp)+t;
        goto[.+2, nocarry];
        xftemp←(xftemp)+1;      * end around carry
        t←lcy[xftemp, 1];       * cycle left one place
        stack←t;                * put back on stack
        task, stack&+1;         * point stack at count
        stack←(stack)+1;                * increment count
        lu←xfwdc;                       * check interrupts enabled
        goto[ChkSNext, alu#0], lu←nenww;* check if interrupts present
        goto[ChkSNext1, alu=0], t←(stack&-1)-1;
* interrupt!!! - copied from xf.mc @BLT
        task, t←(2C);           * pcx needs to be decremented by 2
        nop;
        t← (pcxreg)-(t);        * since it's a MISC bytecode
        goto[.+2, nocarry], xftemp←t;
        mIPC←(mIPC)-(4C);               * point to old code buffer
        loadpage[mIPage4];
        goto[intstop], pcf←xftemp;

ChkSEnd:        t←(zero) -1;    * checksum=-1?
        lu←(stack) XOR t;
        goto[.+2, alu#0];
        stack←(stack)+1;        * make chksum =0
        goto[lpp2nxtret];       * goto NEXTINST, return
%

*SetMaintenancePanel opcode
* just put value on stack
@SetMP: T ← stack&-1, loadpage[0], at[MiscDisp, 10];
        callp[PNIP];
        lu ← NextInst[IBUF], call[MiscTailx];

*=******* Start of Pilot Code *******************************************
*Read realtime clock; Switch to task 17 to read the clock
@RCLK:  RTemp ← (ReadTimeAddr), at[MiscDisp, 11];
        APC&APCTask ← RTemp, call[MiscRet];
        goto[MiscTail];

        SetTask[17];
ReadTime:
        T ← ClockLo, AT[ReadTimeLoc];
        Stack&+1 ← T;
        T ← ClockHi;
        Stack&+1 ← T, return;
        SetTask[0];
*=******* End of Pilot Code *********************************************

*#******* Start of Alto Code *********************************************
*Read realtime clock (push[RM325]; push[VM430]; check for overflow
@RCLK:  T ← (R400) OR (30c), at[MiscDisp, 11];
        Pfetch1[Nova, RTEMP];
        RTEMP1 ← 325c;
        T ← Stkp;
        Stkp ← RTEMP1, RTEMP1 ← T, NoRegILockOK;
        RTEMP1 ← (RTEMP1) xor (377c);
        T ← lsh[Stack,1], skip[R>=0]; * check for non posted overflow
         RTEMP ← (RTEMP) + 1;
        Stkp ← RTEMP1;
        Stack&+1 ← T, loadpage[7];
        T ← RTEMP, gotop[PushTP7];
*#******* End of Alto Code ***********************************************

*Read printer
@RPRINTER:      T ← PRINTER, at[MiscDisp, 12];
        Stack&+1 ← T, goto[MiscTail];

*Write printer
@WPRINTER:      PRINTER ← Stack&-1, goto[MiscTail], at[MiscDisp,13];
```

```
*BitBLT
BitBLT: Stack&-1, LoadPage[bbp2], Opcode[366];
        T ← Stack&+1, goto[MesaBitBLT];


*STARTIO
@STARTIO:
*       T ← stack&-1, LoadPage[EEPage], Opcode[366];
        T ← stack&-1, LoadPage[opPage3], Opcode[366];
        goto[preEESIO], RTEMP1 ← 1c; *RTEMP1 is the return indicator (Mesa/Nova)

*JRAM
@JRAM:  T ← sUnimplemented, goto[kfcr],Opcode[367]; *Nova - flavored JMPRAM

*DST
@DST:   T ← NextData[IBuf] , Opcode[370];
        t ← (LOCAL) + (t) ,loadpage[xfpage1];
        usectask,call[SaveState] , xftemp ← t;
        T ← (xftemp) + (12c);
        Pstore1[MDS, xfMY], goto[P7Tail];

*LST
@LST:   T ← NextData[IBuf], Opcode[371];
        call[SavPCinFrame],xfTemp ← T;* NextData must be before SavePC call
        GOTO[LSTFgo], T ← (xfTemp);

*LSTF
@LSTF:  T ← NextData[IBuf], Opcode[372];
        MemStat ← (FreeFrame);          * Set FreeFlag
LSTFgo: T ← (LOCAL) + (T), LoadPage[xfpage1];
        GOTOp[LoadState], xfTemp ← T;   * xfTemp is pointer to saved state.


*WR
@WR:    T ← NextData[IBuf] , Opcode[374];
        xftemp ← T , loadpage[xfpage1];
        dispatch[xftemp,16,2];

        onpage[xfpage1];
        disp[xfwr] , T ← stack&-1;
xfwr:   goto[MiscTail] , xfwdc ← T ,     at[xfwrtab,1];
        goto[MiscTail] , xfXTSreg ← T , at[xfwrtab,2];
*=******* Start of Pilot Code *******************************************
        MDShi ← T , at[xfwrtab,3];
        T ← MDShi ← (LSH[MDShi,10]) + (T);
        LOCALhi ← T;
        GLOBALhi ← T, goto[MiscTail];
*=******* End of Pilot Code *********************************************

*RR
@RR:    T ← NextData[IBuf] , Opcode[375];
        xftemp ← T , loadpage[xfpage1];
        dispatch[xftemp,15,3];

        onpage[xfpage1];
        LoadPage[4], disp[xfrr];
xfrr:   gotop[PushT], T ← xfwdc ,    at[xfrrtab,1];
        gotop[PushT], T ← xfXTSreg , at[xfrrtab,2];
        gotop[PushT], T ← xfXTPreg , at[xfrrtab,3];
        gotop[PushT], T ← xfATPreg , at[xfrrtab,4];
        gotop[PushT], T ← xfOTPreg , at[xfrrtab,5];
        gotop[PushT], T ← MDShi , at[xfrrtab,6];

*BRK
@BRK:   t ← zero , goto[kfcr] , Opcode[376];


*Cause pagefault trap - done only by fault, is not supposed to be encountered ininstruction stream.
TrapFlap:       T ← (PCFReg)-1, Opcode[377];     * back up PCF by one
*       RTEMP ← T, LoadPage[FaultPage1];
        RTEMP ← T;
        PCF ← RTEMP, gotop[preStartMemTrap];


*Unused opcodes on page 7
        T ← sUnimplemented, goto[kfcr], Opcode[373];

*-----------------------------------------------------------
* Following codes added to provided external referrences

                          OnPage[opPage3];
preEESIO:                 RTEMP ← (1400c);
                RTEMP ← (RTEMP) or (105c);
                APC&APCTask ← RTEMP;
                Return; * goto EESIO of the EtherTask at 1505

preStartMemTrap:          RTEMP ← (6000c);
                RTEMP ← (RTEMP) or (16c);
                APC&APCTask ← RTEMP;
                Return; * goto StartMemTrap of the Fault at 6016


*SUBROUTINE PNIP puts the number in T into the maintenance panel
*It will be used after initialization is complete
*Does not task unless called from task 0
        ONPAGE[0];

PNIP:   usectask, RTEMP ← T, at[PNIPBase,20];
        T ← APC&APCTask, at[PNIPBase,17];
        RCNT ← T, ClearMPanel, call[.+1], at[PNIPBase,0];
```

```
PNloop: RTEMP1 ← 4C, at[PNIPBase,1];
        RTEMP1 ← (RTEMP1)-1, dblgoto[.+1,.,ALU<0], at[PNIPBase,6];
        RTEMP  ← (RTEMP) - 1, at[PNIPBase,7];
        lu ← ldf[RCNT,0,4], goto[PNdone, ALU<0], at[PNIPBase,16];
        skip[alu#0], at[PNIPBase,4];
          IncMPanel, return, at[PNIPBase,2]; * task 0, tasking ok
        IncMPanel, goto[PNloop], at[PNIPBase,3]; * task #0, tasking not allowed
PNdone: APC&APCTask ← RCNT, at[PNIPBase,5];
        return, at[PNIPBase,15];

        end;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
        TITLE[NovaEmulator];

*Last Modified by Johnsson October 9, 1979  4:48 PM, Mesa 6.0 JMPRAM
* Modified by Chang September 7, 1979  4:30 PM, clean page 2
* Modified by Chang August 20, 1979  6:40 PM, move Timer's regs
* Modified by Johnsson June 13, 1979  10:56 AM, remove at's on page 0
* Modified by Johnsson June 11, 1979  11:41 AM, move MulDiv
* Modified by Chang May 27, 1979  2:07 PM, nail down neNoskip
* Modified by Johnsson May 9, 1979  2:28 PM
* Modified March 27, 1979  3:28 PM by Sandman
SET[neBase,lshift[nePage,10]];

        ONPAGE[nePage];
*Dispatch base addresses
SET[neID,ADD[neBase,20]];
SET[neCY,ADD[neBase,40]];
SET[neFunct,ADD[neBase,60]];
SET[neSH0,ADD[neBase,100]];
SET[neSH1,Add[neBase,120]];
SET[neInX,Add[neBase,140]];
Set[Jtab, Add[neBase,160]];
Set[NEXA, Add[neBase,220]];
Set[NEXB, Add[neBase,240]];
Set[NEXC, Add[neBase,260]];
* Set[nePage1, 2];
Set[nePage1, 0];
Set[neNoskipLoc,Add[lshift[nePage,10],273]];
Set[mdPage,11];

*Nova Emulator
*Assumes that PC is in the base register PCB and in PCF.  On entry,
*PCF points to the first (even) byte of an instruction. Odd entry
*points are neSkipx, neNoskip.  Even entries are neSkip, neNoskipx.

*The instruction at location 1 (BufferRefillTrap) is
*'loadpage[0], goto[377]', which sends control to location 377
*on the page that did the (aborted) NextInst/NextData.  We would
*like the instruction at 0 to be a Pfetch4, but alas, DF2 addressing
*doesn't work, since this is the only case in which an aborted
*instruction is not executed immediately, and H2 is not loaded
*in the  cycle following an aborted instruction (so the displacement
*won't be loaded).

NovaRefill:  gotop[.+1], Pfetch4[PCB,IBUF,4], at[neBase,377];

        ONPAGE[0]; *buffer refill for Nova

        PCB ← (PCB) + (4C);
        PCF ← RZero;
IntTest:
        LoadPage[nePage1];
        lu ← NWW, gotop[.+2,R>=0];

        onpage[nePage1];
        return; *interrupts are disabled
        goto[.+2,ALU#0], T ← (R400) or (52C); *start base register setup
        return; *no pending interrupts
        DMA ← T,usectask;
        T ← apc&apctask,task;
        intRTN ← T;
        Pfetch2[DMA,WW,0]; *fetch WW and ACTIVE
        T ← NWW;
        WW ← T ← (WW) or (T);
        T ← (ACTIVE) AND (T);
        GOTO[intT8,ALU#0], RTEMP ← T;
        NWW ← (0C);               *no active interrupts - inactive ones are in WW - clear NWW
        PSTORE1[DMA,WW,0];                  *store WW, TASK
        APC&APCTASK ← intRTN;
intRET: DMA ← 0C, return; *restore the base register


*we are going to start an interrupt
intT8:
        T ← INTX ← (1C) ;          *INTX will contain interrupt mask
        NWW ← (100000C);           *disable interrupts, clear NWW
        CALL[intT8A] , RTEMP1 ← T ;     *RTEMP1 will contain interrupt level index

intT8A:
        RTEMP ← RSH[RTEMP,1], GOTO[intT9,R ODD]; *loop to get number of the highest priority interrupt
        T ← INTX ← LSH[INTX,1] ;
        RTEMP1 ← (RTEMP1) + 1, RETURN ;

intT9:
        WW ← (WW) AND NOT (T);          *enter int routine - save other interrupts in WW
        PSTORE1[DMA,WW,0],task; *store WW at 452b, TASK
        T ← (ldf[GETRSPEC[127],14,3]); *recover the PC
        PCB ← (PCB) + (T);
        lu ← intRTN, goto[.+2,Rodd]; *intRTN even means we got here from neSkip, and
*we must increment the PC before saving it away
        PCB ← (PCB) + 1;
        T ← (R400) OR (100C) ;
        PSTORE1[NOVA,PCB], call[intRET];        *save PC at 500b, TASK
        T ← (RTEMP1) + (T),loadpage[nePage];    *T ← address of new PC
        gotop[BRIx];

        ONPAGE[nePage];

neSkip: FF1@[17], call[eSkip]; *neSkip is even
```

```
neNoskip:       T← CNextData[IBUF], call[neS1], at[neNoskipLoc]; *neNoskip is odd
        Dispatch[RTEMP,10,3], goto[NESx]; *dispatch on first 3 bits of opcode

neSkipx:        goto[neSkip]; *neSkipx is odd
neNoskipx: goto[neNoSkip]; *neNoskipx is even

eSKip:  FF1@[17], goto[neNoskip]; *can't cause buffer refill
```

```
*SUBROUTINE nes1 does setup for 2Acc instructions, calculates effective address
*for 1Acc instructions.  Returns R address of AC0 in T.

neS1:    RTEMP ← T,dblgoto[ne2Acc,ne1Acc,H2bit8];

ne2Acc: PCF[IBUF] ← rcy[PCF[IBUF],3]; *put NoLoad bit into bit 15
neS1ret:        T ← pAC0, return; *pointer to AC0

ne1Acc: Dispatch[PCF[IBUF],5,4]; *get Effective Address - dispatch on I,X,and disp.0 (16 way)
        Disp[InX0], T ← rhmask[PCF[IBUF]];

InX0:   EfAd ← T, goto[neS1ret], AT[neInX,0]; *Page 0
InX1:   EfAd ← T, goto[neS1ret], AT[neInX,1];

InX2:   T ← (ldf[GETRSPEC[127],15,2]) + (T), AT[neInX,2]; *PC relative
        T ← (PCB) + (T), goto[InX0];
InX3:   T ← (PCF[IBUF]) or (177400C), goto[InX2], AT[neInX,3];

InX4:   T ← (AC2) + (T), goto[InX0], AT[neInX,4]; *AC2 relative
Inx5:   T ← (PCF[IBUF]) or (177400C), goto[InX4], AT[neInX,5];

InX6:   T ← (AC3) + (T), goto[InX0], AT[neInX,6]; *AC3 relative
Inx7:   T ← (PCF[IBUF]) or (177400C), goto[InX6], AT[neInX,7];

InX10:  Pfetch1[Nova,EfAd],goto[neS1ret], AT[neInX,10]; * Page 0 indirect
InX11:  Pfetch1[Nova,EfAd],goto[neS1ret], AT[neInX,11];

InX12:  T ← (ldf[GETRSPEC[127],15,2]) + (T), AT[neInX,12];      *PC relative indirect
        T ← (PCB) + (T), goto[InX10];
InX13:  T ← (PCF[IBUF]) or (177400C), goto[InX12], AT[neInX,13];

InX14:  T ← (AC2) + (T), goto[InX10], AT[neInX,14]; *AC2 relative indirect
Inx15:  T ← (PCF[IBUF]) or (177400C), goto[InX14], AT[neInX,15];

InX16:  T ← (AC3) + (T), goto[InX10], AT[neInX,16]; *AC3 relative indirect
Inx17:  T ← (PCF[IBUF]) or (177400C), goto[InX16], AT[neInX,17];
```

```
NESx:   Disp[neID0], RTEMP ← (ldf[RTEMP,13,2]) + (T);

*Main instruction dispatch
neID0:  Dispatch[RTEMP,16,2], goto[Jdisp], AT[neID,0];  *jumps
neID1:  STKP ← RTEMP, goto[LDA], AT[neID,1];    *lda
neID2:  STKP ← RTEMP, goto[STA], AT[neID,2];    *sta
neID3:  lu ← ldf[PCF[IBUF],3,1], goto[XOPA], AT[neID,3];        *extended ops
neID4:  T ← AC0, goto[getCY], AT[neID,4]; *arith(AC0)
neID5:  T ← AC1, goto[getCY], AT[neID,5]; *arith(AC1)
neID6:  T ← AC2, goto[getCY], AT[neID,6]; *arith(AC2)
neID7:  T ← AC3, goto[getCY], AT[neID,7]; *arith(AC3)

LDA:    Stack&-1, FF1@[17], task; *Since fetches to the stack increment STKP
        T ← EfAd;
        Pfetch1[Nova,Stack], goto[neNoskip];

STA:    T ← EfAd;
        PStore1[Nova,Stack], call[xnPG2RET];
        F1@[17], goto[neNoskip];

*Jmp,JSR,ISZ,DSZ - enter with dispatch on function pending
Jdisp:  Disp[Jmp], T ← EfAd;

Jmp:    Pfetch4[ Nova, IBUF], AT[Jtab,0];
Jmpx:   PCB ← T, T ← T, call[Jtsk]; *bypass kludge
        T ← CNextData[IBUF], call[neS1], at[neBase,15]; *cannot cause IBUF refill (but must be
*in an odd location, since it is the return from IntTest (via Jtsk).
        Dispatch[RTEMP,10,3], goto[NESx];

*SUBROUTINE Jtsk Updates PCB and PCF, then tests for interrupts
Jtsk:   PCB ← lsh[PCB,1];
        PCF ← PCB, PCB ← T;
        loadpage[0];
        PCB ← (PCB) and not (3C), goto[IntTest];

JSR:    PFetch4[Nova, IBUF], AT[Jtab,1];
        T ← PCB, call[GPC1]; *recover the PC
        AC3 ← T;
        T ← EfAd, goto[Jmpx];

ISZ:    Pfetch4[Nova, xBuf], AT[Jtab,2];
        EfAd ← lcy[Efad,4], call[neXsetup];
        DB[xBuf] ← (DB[xBuf]) + 1,  goto[DSZx];

DSZ:    Pfetch4[Nova, xBuf], AT[Jtab,3];
        EfAd ← lcy[Efad,4], call[neXsetup];
        DB[xBuf] ← (DB[xBuf]) - 1;
DSZx:   FF1@[17], FREEZERESULT; *advance PC, let DB be written
        PStore4[Nova, xBuf], dblgoto[neSkip, neNoskip, ALU=0];

*Set up DB to point to the register addressed by the low two bits of EfAd.
*Call with: T ← lcy[EfAd, 4], call[neXsetup];
neXsetup:       DB ← EfAd;
        BBFBx, return; *advance DB to DBX

GPC1:   T ← (ldf[GETRSPEC[127],14,3]) + (T) +1, return;
```

```
*2 Accumulator instructions.  ACS is in T
getCY:  Dispatch[PCF[IBUF],15,2];        *dispatch on carry field of instruction
        Disp[neCY0], Stkp ← RTEMP;       *load index and dispatch

neCY0:  Dispatch[PCF[IBUF],10,3], AT[neCY,0]; *Cin ← CARRY (in bit 0 of Carry)
        Disp[neFunct0], lu ← Carry;
neCY1:  Dispatch[PCF[IBUF],10,3], AT[neCY,1]; *Cin ← 0
        Disp[neFunct0], lu ← 0c;
neCY2:  Dispatch[PCF[IBUF],10,3], AT[neCY,2]; *Cin ← 1
        Disp[neFunct0], lu ← 100000c;
neCY3:  Dispatch[PCF[IBUF],10,3], AT[neCY,3]; *Cin ← CARRY'
        Disp[neFunct0], lu ← (Carry) xor (100000c);

neFunct0:       T ← (Zero) xnor (T), dblgoto[ntc1,ntc0,ALU<0], AT[neFunct,0]; *com
neFunct1:       T ← (Zero) - (T), dblgoto[tc1,tc0,ALU<0], AT[neFunct,1]; *neg
neFunct2:       dblgoto[ntc1,ntc0,ALU<0], AT[neFunct,2]; *mov
neFunct3:       T ← (Zero) + (T) +1, dblgoto[tc1,tc0,ALU<0], AT[neFunct,3]; *inc
neFunct4:       T ← (Stack) - (T) -1, dblgoto[tc1,tc0,ALU<0], AT[neFunct,4]; *adc
neFunct5:       T ← (Stack) - (T), dblgoto[tc1,tc0,ALU<0], AT[neFunct,5]; *sub
neFunct6:       T ← (Stack) + (T), dblgoto[tc1,tc0,ALU<0], AT[neFunct,6]; *add
neFunct7:       T ← (Stack) and (T), dblgoto[ntc1,ntc0,ALU<0], AT[neFunct,7]; *and

*ALUCY has no effect on carry - dispatch on shift field
ntc0:   Dispatch[PCF[IBUF],13,2], goto[Cout0];
ntc1:   Dispatch[PCF[IBUF],13,2], goto[Cout1];

*ALUCY complements incoming carry - dispatch on shift field
tc0:    Dispatch[PCF[IBUF],13,2], dblgoto[Cout1,Cout0,Carry]; *incoming carry was 0
tc1:    Dispatch[PCF[IBUF],13,2], dblgoto[Cout0x,Cout1x,Carry]; *incoming carry was 1

Cout0:  Disp[neSH00], Result ← T;
Cout1:  Disp[neSH10], Result ← T;

Cout0x: Disp[neSH00], Result ← T;
Cout1x: Disp[neSH10], Result ← T;

*Shift dispatch for final carry = 0
neSH00: PCF[IBUF], lu ← T, dblgoto[noload0, load0, Rodd], AT[neSH0,0]; *no shift
neSH01: T ← Result ← lsh[Result,1], AT[neSH0,1], dblgoto[fc1,fc0,R<0]; *left shift
neSH02: T ← Result ← rsh[Result,1], AT[neSH0,2], dblgoto[fc1,fc0,Rodd]; *right shift
neSH03: T ← Result ← lcy[Result,10], AT[neSH0,3], goto[fc0]; *swap

*Shift dispatch for final carry = 1
neSH10: PCF[IBUF], lu ← T, dblgoto[noload1, load1, Rodd], AT[neSH1,0]; *no shift
neSH11: T ← Result ← (lsh[Result,1]) + 1, AT[neSH1,1], dblgoto[fc1,fc0,R<0]; *left shift
neSH12: Result ← rcy[Result,1], AT[neSH1,2]; *right shift
        T ← Result ← (Result) or (100000C), dblgoto[fc1,fc0,R<0];
neSH13: T ← Result ← lcy[Result,10], AT[neSH1,3], goto[fc1]; *swap

fc0:    PCF[IBUF], lu ← T, dblgoto[noload0, load0, Rodd]; *test NoLoad, set to test Result=0
fc1:    PCF[IBUF], lu ← T, dblgoto[noload1, load1, Rodd];

noload0:        T ← Stack, dblgoto[cZrZ,cZrN,ALU=0]; *T ← original ACD
load0:  Carry ← 0C, dblgoto[cZrZ,cZrN,ALU=0];

noload1:        T ← Stack, dblgoto[cNrZ,cNrN,ALU=0];
load1:  Carry ← 100000C, dblgoto[cNrZ,cNrN,ALU=0];

cZrZ:   PCF[IBUF] ← (PCF[IBUF]) and (160000C), goto[tsk];
cZrN:   PCF[IBUF] ← (PCF[IBUF]) and (60000C), goto[tsk];
cNrZ:   PCF[IBUF] ← (PCF[IBUF]) and (120000C), goto[tsk];
cNrN:   PCF[IBUF] ← (PCF[IBUF]) and (20000C), goto[tsk];

tsk:    Stack ← T, call[neS2];  *TASK at last
        lu ← ldf[PCF[IBUF],15,2], dblgoto[tsf,tsb,Rodd];

neS2:   PCF[IBUF] ← ldf[PCF[IBUF],0,3], return;

*Test skip, point PCF at first byte of next instruction (cannot cause refill)
tsf:    dblgoto[neSkip,neNoskip,ALU=0], F1@[17];
tsb:    dblgoto[neSkipx,neNoskipx,ALU#0], F1@[17];
```

```
*               NOVA AUGMENTED INSTRUCTION SET

*       60000           CYCLE   Left Rotate
*       61000           DIR     Disable interrupts
*       61001           EIR     Enable interrupts
*       61002           BRI     Branch and return from interrupt
*       61003           RCLK    Read Clock
*       61004           SIO     Start I/O
*       61005           BLT     Block Transfer
*       61006           BLKS    Block Store
*       61007           **SIT   Start Interval Timer - returns -1
*       61010           JMPRAM  Jump to RAM - only works if AC1 = 420 (enters Mesa)
*       61011           RDRAM   Read RAM - returns -1
*       61012           WRTRAM  Write RAM - NOP
*       61013           DIRS    Disable interrupts and skip if on
*       61014           VERS    Version - AC0 ← 40000C
*       61015           **DREAD Double-word read (altoII only)
*       61016           **DWRITE        Double-word write (altoII only)
*       61017           **DEXCH Double-word exchange (altoII only)
*       61020           MUL     Unsigned Multiply
*       61021           DIV     Unsigned Divide
*       61022           **DIAGNOSE1     Diagnostic (altoII only)
*       61023           **DIAGNOSE2     Diagnostic (altoII only)
*       61024           BITBLT  Bit Block Transfer
*       64400           JSRII   Jump to subroutine, double indirect, pc relative
*       65000           JSRIS   Jump to subroutine, double indirect, ac2 relative
*       67000           CONVERT Scan conversion of characters

*       NOTE: instructions with ** are not implemented, and will bomb out if executed

*Dispatch to 8 main extended opcodes. Enter with ALU = Instruction.3.
XOPA:   Dispatch[PCF[IBUF],4,3], goto[xnUNIMPTRAP,ALU#0]; *70000 - 77777 are unimplemented
        Disp[Cycle], T ← EfAd;
```

```
*Left cycle AC0 by inst[12-15d], or by AC1 if count is 0.

Cycle:  RTEMP ← 17C, goto[.+3,ALU#0],AT[NEXA,0];        *cycle - Test EfAd for 0
        T ← AC1;*cycle by AC1 if count=0
        EfAd ← T;
        EfAd ← (Efad) -1;
        RTEMP ← (RTEMP) - (T);
        CycleControl ← EfAd, TASK;
        T ← RF[AC0];
        CycleControl ← RTEMP, goto[.+2,R<0];
        T ← (WFA[AC0]) or (T);
        AC0 ← T, F1@[17], goto[neNoskip];

*Opcodes 61000 - 61024
XOP1:   lu ← ldf[PCF[IBUF],13,1], AT[NEXA,1]; *test bit 11d
        DISPATCH[PCF[IBUF],14,4], DBLGOTO[xnXA1A,xnXA1B,ALU=0]; *dispatch on bits 12-15d
xnXA1A: DISP[DIR];
xnXA1B: DISP[MUL];

*Opcode 62000 - unused
*xnXA2: GOTO[xnUNIMPTRAP], AT[NEXA,2] ;

*Opcode 63000 - unused
*xnXA3: GOTO[xnUNIMPTRAP], AT[NEXA,3] ;

*JSRII - 64400
JSRII:  PFETCH1[Nova,EfAd], AT[NEXA,4];
xnJSRI: call[xnPG2RET];
        T ← EfAd;
        PFETCH1[Nova,EfAd], call[xnPG2RET];
        T ← EfAd, goto[JSR];

*JSRIS - 65000
JSRIS:  PFETCH1[Nova,EfAd],GOTO[xnJSRI], AT[NEXA,5];

*Opcode 66000 - unused
*xnXA6: GOTO[xnUNIMPTRAP], AT[NEXA,6] ;
```

```
*CONVERT - Opcode 670xx
*Registers:
*       ac0:    destination word address minus NWRDS
*       ac2     + disp points to two word block:
*                       word 0  NWRDS -- number of words per scanline (< 400c)
*                       word 1  dba -- minus dest bit addr mod 20c
*       ac3:    pointer to word xh of the character descriptor block
*
*Character Descriptor Block:
*       words 0:        xh-1    bit map for character
*       word xh:        xw -- (2*width) + 1, or (2*pseudochar)  if extension required
*       word xh+1:      hd,,xh -- (scan lines to skip),,(height of bit map)
*NOTE:  This instruction looks like an @AC2 instruction, so at entry EfAd contains the first word
*of the 2-word block above (i.e. NWRDS).
*The high halves of base registers SMA and DMA are assumed to be set up.

CONVERT:        T ← ldf[PCF[IBUF],10,10],AT[NEXA,7] ;   *displacement
        T ← (AC2) + (T) + 1;
        PFETCH1[Nova,AC1], call[AC3toT];                *fetch dba
        PFETCH1[Nova,RTEMP] ;   *fetch self-relative pointer to xw
        T←AC0,TASK;
        DMA ← T, FF1@[17];      *setup  for later, advance PC
        T←RTEMP; *self-relative pointer to xw
        T←AC3←(AC3)+(T)+1; *add pointer
        PFETCH1[Nova,xnXH];     *fetch hd,,xh
        AC1 ← T ←  (AC1) AND (17C), TASK;       *mask dba
        RTEMP ← 16C;
        RTEMP ← (RTEMP) - (T);
        T←EfAd,CALL[FIXDMA]; *DMA←DMA + (hd*xnNWRDS)

FIXDMA: xnXH←(xnXH)+(177400C);
        DMA←(DMA)+(T),GOTO[DMAfixed,ALU<0]; *when done, DMA will point to the first dest word
xnPG2RET:       RETURN;

DMAfixed:       AC3←(AC3)-(2C); *back up AC3 to the start of the block (AC3 ← AC3 - xh -2)
        xnXH←T←(LDF[xnXH,10,10])-1; *also decrement xh
        T←(AC3)-(T)-1,TASK;
        SMA←T;


xnCVLOOP:       SMA←T←(SMA)+1;

        PFETCH1[Nova,AC3], task; *fetch source
        XBI ← pXBuf;
        xnXH←(xnXH)-1,GOTO[xnCNVEND,R<0]; *test count
        PFETCH4[DMA,xBuf,0]; *fetch dest
        T ← (ldf[DMA,16,2]), task; *use Stkp to index xBuf
        XBI ← (XBI) + (T);
        Stkp ← XBI;
        CycleControl ← AC1;
        T ← (RF[AC3]), task; *source contribution to first dest word
        Stack ← (Stack) or (T);
        CycleControl ← RTEMP, goto[.+2,R>=0];
        PStore4[DMA,xBuf,0], goto[CVXx]; *source exhausted
        T ← WFA[AC3];   *source contribution to second dest word
        XBI ← (XBI) + 1, goto[xnCVX, ALU=0]; *if it is zero, we are done
        lu ← ldf[XBI,16,2]; *check whether we are still in the quadword
        Stkp ← XBI, goto[doSingleWord,ALU=0];
        Stack ← (Stack) or (T), goto[xnCVx]; *OR the second Dest word

doSingleWord: Pfetch1[DMA, xnDEST, 1]; *fetch second dest word
        call[xnPG2RET];
        xnDEST ← (xnDEST) or (T); *so xnDEST will be written
        Pstore1[DMA, xnDEST,1],call[xnPG2RET]; *UGH
        nop; *allocation constraint

xnCVX:  PStore4[DMA,xBuf,0]; *store buffer
CVXx:   T ← EfAd; *get NWRDS
        DMA←(DMA)+(T), GOTO[xnCVLOOP];

xnCNVEND:       AC3 ← RSH[AC3,1], dblgoto[NeSkipx,neNoskipx,Rodd];
AC3toT: T ← AC3, return;


%
xnCVLOOP:       SMA←T←(SMA)+1;

        PFETCH1[Nova,AC3],CALL[xnPG2RET]; *fetch source
        xnXH←(xnXH)-1,GOTO[xnCNVEND,R<0]; *test count
        PFETCH1[DMA,xnDEST,0]; *fetch dest0
        CycleControl ← AC1,task;
        T ← RF[AC3]; *source contribution to first dest word
        xnDEST ← (xnDEST) or (T);
        Pstore1[DMA, xnDEST,0];
        CycleControl ← RTEMP, goto[.+2,R>=0];
        T ← EfAd, goto[CVXx];
        T ← WFA[AC3];   *source contribution to second dest word
        goto[xnCVX, ALU=0];
        Pfetch1[DMA, xnDEST, 1], call[xnPG2RET]; *fetch second dest word
        xnDEST ← (xnDEST) or (T);
        Pstore1[DMA, xnDEST,1];
xnCVX:  T←EfAd; *get NWRDS
CVXx:   DMA←(DMA)+(T), GOTO[xnCVLOOP];

xnCNVEND:       AC3 ← RSH[AC3,1], dblgoto[NeSkipx,neNoskipx,Rodd];
AC3toT: T ← AC3, return;
%
```

```
*Extended Opcodes with no displacement or parameter
DIR:    NWW ← (NWW) OR (100000C),goto[ComRet], AT[NEXB,0] ;      *dir - 61000

EIR:    T ← (R400) OR (52C),goto[EIRcom], AT[NEXB,1] ;  *eir - 61001
EIRcom: PFETCH1[Nova,WW],CALL[xnPG2RET];
        T ← WW;
        NWW ← (NWW) OR (T);
        NWW ← (NWW) AND NOT (100000C);
        PCF[IBUF], dblgoto[EIRz,BRIy,Rodd];
EIRz:   T ← PCB, call[GPC1]; *EIR must test for interrupts NOW.  We simulate JMP .+1
        T ← T, goto[JMP];


BRI:    T ← (R400) OR (52C), goto[EIRcom], AT[NEXB,2] ; *bri - 61002
BRIy:   T ← (R400) OR (100C); *fetch PC from location 500
BRIx:   PFETCH1[Nova,EfAd], CALL[xnPG2RET];
        T ← EfAd, goto[JMP];


RCLK:   T ← (R400) OR (30C), AT[NEXB,3] ;        *rclk - 61003
        PFETCH1[Nova,AC0];
*       RTEMP ← (325C),TASK; *Get RTCLOW from task 16's R's
        RTEMP ← (355C),TASK; *Get RTCLOW from task 16's R's
        STKP ← RTEMP;
        T ← LDF[STACK,1,12],GOTO[xnNRDOVF,R>=0];

*the low bits overflowed, but the display
*hadn't gotten around to updating the high bits yet.
        AC0 ← (AC0)+1;

xnNRDOVF:       AC1 ← T, goto[neNoskip],FF1@[17];

SIO:    T ← AC0, loadpage[opPage3], AT[NEXB,4]; *sio
        RTEMP1 ← 0c, gotop[preEESIO]; *RTEMP1=0 means return to Nova
```

```
*BLT and BLKS
*slow straight forward blt with no checking for source and dest overlap
*       registers:
*       rac0    address of first source word - 1 (BLT), or data to be stored (BLKS)
*       rac1    address of last destination word
*       rac2    unused
*       rac3    negative word count

*BLT - 61005
BLT:    AC0 ← T ← (AC0) +1, AT[NEXB,5]; *blt
BLTx:   Pfetch1[Nova, RTEMP], call[xnPG2RET]; *fetch source
Bloop:  lu ← AC3;
        AC3 ← T ← (AC3) + 1, goto[BLdone, ALU=0]; *test count = 0 and increment it
        T ← (AC1) + (T), loadpage[0]; *form destination address
        Pstore1[Nova,RTEMP], callp[IntTest]; *the return link must be odd, so that if an interrupt
*occurs from IntTest, the saved PC will not be incremented...
        PCF[IBUF], dblgoto[BLTxx,BLKSxx,Rodd], at[neBase,7]; *thus, this location must be odd
BLTxx:  AC0 ← T ← (AC0) + 1, goto[BLTx];
BLKSxx: T ← AC0, goto[BLKSx];

BLdone: AC3 ← 0C, goto[ComRet];
*BLdone:        AC3 ← 0C, goto[STAq]; *STAq refills IBUF and advances PC


*BLKS - 61006
BLKS:   T ← AC0, AT[NEXB,6]; *blks
BLKSx:  RTEMP ← T, goto[Bloop];

*SIT - 61007
*SIT:   GOTO[xnUNIMPTRAP], AT[NEXB,7] ; *sit

JMPRAM: T ← 20c , AT[NEXB,10] ; *jmpram - 61010
        lu ← (rhmask[AC1]) xor (T) ;
        T ← AC0, goto[NEtoMesa,alu=0] ;
        GOTO[xnUNIMPTRAP] ;

NEtoMesa:
        GLOBALhi ← 0C; *Initialize some Mesa Emulator registers
        LOCALhi ← 0c;
        MEMSTAT ← 0c;
        xfXTSReg ← 0c;
        MDS ← 0c;
        xfMX ← 1c;
        TickCount ← 3c;
        xfTemp ← T, LoadPage[7];
        MDShi ← 0c, gotop[MStart];


ComRet: GOTO[neNoskip], FF1@[17]; *common ending for instructions which advances PC

RDRAM:  AC0 ← (ZERO) - 1 , GOTO[ComRet], AT[NEXB,11];    *rdram - 61011

WRTRAM: GOTO[ComRet], AT[NEXB,12];       *wrtram - 61012 (nop)

*dirs - 61013
DIRS:   FF1@[17],AT[NEXB,13] ;
        NWW ← (NWW) or (100000C), dblgoto[neNoskip,neSkip,R<0];

VERS:   AC0 ← (40000c), GOTO[ComRet], AT[NEXB,14] ;      *vers - 61014

*DREAD: GOTO[xnUNIMPTRAP], AT[NEXB,15]; *dread

*DWRITE:        GOTO[xnUNIMPTRAP], AT[NEXB,16]; *dwrite

*DEXCH: GOTO[xnUNIMPTRAP], AT[NEXB,17]; *dexch
```

```
*AC0,,AC1 ← (AC1 * AC2) + (AC0)
MUL:    loadpage[mdPage], AT[NEXC,0];               *mul
        callp[Multiply], T ← 16C;
        GOTO[neNoskip],FF1@[17];

        ONPAGE[mdPage];

*Steps:
*1      test next mpr bit (AC1[17]), rsh[AC1]
*2      add t to AC0
*3      test carry
*4      test AC0[17], rsh[AC0]
*5      or one into high bit of AC1
*6      or one into high bit of AC0
*7      test count
*8      continue
*9      finish

Multiply:
        USECTASK ,RCNT ← T;
        T ← APC&APCTASK ;
        RTEMP1 ← T, call[UMUL8];          *save return address
UMUL1:  dblgoto[UMUL4a,UMUL2,Reven] , AC1 ← rsh[AC1,1] ;

UMUL4a: dblgoto[UMUL7,UMUL5a,r even] , AC0 ← rsh[AC0,1] ;

UMUL2:  goto[UMUL3] , AC0 ← (AC0) + (T) ;
UMUL3:  dblgoto[UMUL4b,UMUL4c,nocarry] ;

UMUL4b: dblgoto[UMUL7,UMUL5a,r even] , AC0 ← rsh[AC0,1] ;

UMUL7:  dblgoto[UMUL9,UMUL8, R<0] , RCNT ← (RCNT) - 1 ;
UMUL5a: goto[UMUL7] , AC1 ← (AC1) or (100000c) ;

UMUL4c: dblgoto[UMUL6,UMUL5b,r even] , AC0 ← rsh[AC0,1] ;

UMUL6:  goto[UMUL7] , AC0 ← (AC0) or (100000c) ;
UMUL5b: goto[UMUL6] , AC1 ← (AC1) or (100000c) ;


UMUL9:  APC&APCTASK ← RTEMP1;
UMUL8:  T ← AC2, RETURN;
```

```
*AC0,,AC1/AC2.  Quotient in AC1, remainder in AC0
DIV:    loadpage[mdPage],T ← AC2, AT[NEXC,1];
        callp[Divide], LU ← (AC0) - (T);                    *div
        FF1@[17],lu ← RCNT,dblgoto[neSkip,neNoskip,Reven]; *Advance PC

        ONPAGE[mdPage];
Divide: dblgoto[noDiv,doDiv,carry],RCNT ← 17C;
noDiv:  return; *RCNT odd means no skip

doDiv:  usectask;
        T ← apc&apctask;
        RTEMP1 ← T; *save link
        T ← AC2, call[UDIVenter];

        lu ← rcy[Carry,1]; *loop RETURNs to here
        AC0 ← (AC0) - (T), dblgoto[UDIV1t,UDIV1f,ALU<0]; *test flag

UDIV1t: AC1 ← (lsh[AC1,1]) + 1, dblgoto[UDIV3t,UDIV3f,R<0]; *no need to test carry, quot ← 1
UDIV1f: dblgoto[UDIV2t,UDIV2f,carry]; *subtract ok, test carry

UDIV2t: AC1 ← (lsh[AC1,1]) + 1, dblgoto[UDIV3t,UDIV3f,R<0]; *put a one in the quotient
UDIV2f: AC0 ← (AC0) + (T); *no carry - undo subtraction
UDIVenter:      AC1 ← (lsh[AC1,1]), dblgoto[UDIV3t,UDIV3f,R<0]; *put a zero in the quotient

UDIV3t: RCNT ← (RCNT) -1, goto[UDIVdone1, R<0];
        AC0 ← (lsh[AC0,1]) + 1, dblgoto[UDIV4t,UDIV4f,R<0];

UDIV3f: RCNT ← (RCNT) -1, goto[UDIVdone2, R<0];
        AC0 ← (lsh[AC0,1]), dblgoto[UDIV4t,UDIV4f,R<0];

UDIV4t: Carry ← (Carry) or (1c), return; *set flag
UDIV4f: Carry ← (Carry) and not (1c), return; *clear flag

UDIVdone1:      APC&APCTask ← RTEMP1, goto[UDIV4f];
UDIVdone2:      APC&APCTask ← RTEMP1, goto[UDIV4f];

        ONPAGE[nePage];
```

```
*DIAGNOSE1:
*xnXC02:        GOTO[xnUNIMPTRAP], AT[NEXC,2] ;*diagnose1

*DIAGNOSE2:
*xnXC03:        GOTO[xnUNIMPTRAP], AT[NEXC,3] ;*diagnose2

BITBLT: T ← Novah, LoadPage[bbp2], AT[NEXC,4];
        AC3 ← T, gotop[NovaBitBLT];

*xnXC05:        GOTO[xnUNIMPTRAP], AT[NEXC,5] ;
*xnXC06:        GOTO[xnUNIMPTRAP], AT[NEXC,6] ;
*xnXC07:        GOTO[xnUNIMPTRAP], AT[NEXC,7] ;
*xnXC10:        GOTO[xnUNIMPTRAP], AT[NEXC,10] ;
*xnXC11:        GOTO[xnUNIMPTRAP], AT[NEXC,11] ;
*xnXC12:        GOTO[xnUNIMPTRAP], AT[NEXC,12] ;
*xnXC13:        GOTO[xnUNIMPTRAP], AT[NEXC,13] ;
*xnXC14:        GOTO[xnUNIMPTRAP], AT[NEXC,14] ;
*xnXC15:        GOTO[xnUNIMPTRAP], AT[NEXC,15] ;
*xnXC16:        GOTO[xnUNIMPTRAP], AT[NEXC,16] ;
*xnXC17:        GOTO[xnUNIMPTRAP], AT[NEXC,17] ;

*       save the current pc + 1 in memory location 527b
*       and jump to location pointed to by location 530b + inst[3,7]

xnUNIMPTRAP:
        T ← (ldf[GETRSPEC[127],14,3]) +1;
        PCB ← (PCB) + T ;
        T ← (R400) OR (127C) ;
        TASK , PSTORE1[Nova,PCB] ;
        T ← (R400) OR (130C) ;
        T ← (ldf[PCF[IBUF],3,5]) + (T), goto[BRIx];

        END;
```

```
* Standard preamble for xxxOccupied.mc files produced by MicroD
* last edited April 23, 1979  3:41 PM

        Insert[D0Lang];
        NoMidasInit;LangVersion;MultDIB;
```

```
BUILTIN[insert,24];
insert[d0lang];
NOMIDASINIT;
LANGVERSION;
MULTDIB;

insert[GlobalDefs]; *task and page assignments

TITLE[Overlay];

*last edit by Chang, August 13, 1979  3:07 PM, RDC Integration
* edit by Johnsson, June 15, 1979  3:24 PM, new registers
*edit by Chang, June 4, 1979  2:09 PM, for MIDAS Overlay booting
*edit by Chang, May 27, 1979  6:39 PM, for Overlay booting
*edit by Johnsson, May 15, 1979  3:27 PM, PNIP fix
*edit by Chang, May 10, 1979  8:31 AM, add new Ethernet ID
*edit by Sandman, April 6, 1979  3:50 PM

*Modified March 6, 1979 by CPT. Added fault handling

RV[rlink0,34]; *subroutine return link

*Registers for other sections of initialization
RV[xCNT,20]; *used everywhere
RV[DevIndex,21]; *used in DeviceInit
RV[contemp,22]; *used in DeviceInit
RV[initr0,40]; *used in DiskBoot
RV[initr1,41]; *used in DiskBoot
RV[initr2,42]; *used in DiskBoot
RV[initr3,43]; *used in DiskBoot
RV[ErrorCnt,44]; *used in DiskBoot
RV[ErrorCountx,45]; *used in DiskBoot
RV[temp,51];


INSERT[DMdefs];

SetTask[0];
* MC[NextDiskAddr,237];
RV[BootDiskAddr,37]; *RDC Integration
MC[CSMemStart,4000];     * temporary memory starting address for CS image
RV[MemAddr,51];          * current address for CS image
RV[VersionID,52];

*Maintenance Panel Normal Operation Codes:
MC[StartDiskBoot,170]; *120d
MC[SystemRunning,202]; *130d

*Maintenance Panel Failure Codes:
MC[NotEnoughMemory,145]; *101d
MC[BadMap,146]; *102d
MC[NoDiskStatus,171]; *121d
MC[BadBoot,172]; *122d

SETTASK[0];
                OnPage[InitPage];

BootSecondBlock:        LoadPage[LoadCSPage];
        gotop[BootSBlock];

                OnPage[LoadCSPage];
BootSBlock:     xCNT ← FFaultAdd;
        stkp←xCNT;
        LU ← (stack) and (40000c);*check Midas Present bit in FFAULT
        goto[BootLoader, alu#0];
        MemAddr ← CSMemStart;
        * Skip VersionID at location CSMemStart (4000)
*       ClearMPanel; *********************** remove this word after testing
*       xCNT←NextDiskAddr;
*       stkp←xCNT;
*       t←stack;                    *get next disk address from reg 237
        t ← BootDiskAddr;           *set next disk address from reg 37
        xCNT←t;
        DMA←(1000c);                *dcb address
        pstore1[DMA,xCNT,IOCBDiskAdr!],call[LoadCSRet];*next disk address

readnextblock:
*       IncMPanel; *********************** remove this word after testing
        ErrorCnt←(10c);
setdcb:
        DevIndex ← (zero) - 1 ;
        initr0 ← 0c;
        initr1←0c;
        initr2←0c;
        pstore2[DMA,initr0,IOCBNext],call[LoadCSRet];
        initr3←0c;
        t←(R400)or(120c);
        pstore4[Nova,initr0],call[LoadCSRet];*clear disk communication cells
        xCNT←(zero) ;   *disk status
        pstore1[DMA,xCNT,IOCBStatus],call[LoadCSRet];
        xCNT←(44000c);  *disk command
        pstore1[DMA,xCNT,IOCBCommand],call[LoadCSRet];
        xCNT←(1400c);   *header address
        pstore1[DMA,xCNT,IOCBHeaderPointer],call[LoadCSRet];
        xCNT←(2000c);   *label address
        pstore1[DMA,xCNT,IOCBLabelPointer],call[LoadCSRet];
        xCNT←(3000c);   *data address
        pstore1[DMA,xCNT,IOCBDataPointer],call[LoadCSRet];

*       start the disk
startdisk:
```

```
        initr1←1000c; * word 521 -- IOCB pointer
        initr3←(zero) - 1; * word 523 = -1 to force a seek
        t←(R400)+(120c); * store into 520-523 (words 0 and 2 are zero)
        pstore4[Nova,initr0], call[LoadCSRet];  *dcb address

*       wait for completion
        ErrorCountx ← 20c;
SetDW:  DevIndex ← (zero)-1; *loop count for status wait
WaitDisk:       Pfetch1[DMA,xCNT,1]; *fetch status word at 1001b
        T ← 17C, call[LoadCSRet];
        lu ← (ldf[xCNT,4,4]) xor (T);
        goto[StatusHere,ALU=0], lu ← ldf[xCNT,10,10];
        DevIndex ← (DevIndex)-1;
        goto[WaitDisk,ALU#0];
        ErrorCountx ← (ErrorCountx) -1;
        goto[SetDW,ALU>=0];
        LoadPage[InitPage];
        T ← NoDiskStatus, gotop[InitFail]; *timed out waiting for disk to store status

StatusHere:  dblgoto[NoFaultStatus,IncECount,ALU=0],FREEZERESULT;

NoFaultStatus:  nop;
        call[loadmi];
        goto[readnextblock];

IncECount:
        ErrorCnt ← (ErrorCnt)-1;
        goto[FailBoot,ALU < 0]; *bad header, try again
        goto[BootSBlock];

FailBoot:               LoadPage[InitPage];
        T ← BadBoot, gotop[InitFail];

                OnPage[InitPage];
LoadOtherCS:    LoadPage[LoadCSPage];
        xCNT ← FFaultAdd, gotop[.+1];

                OnPage[LoadCSPage];
        stkp ← xCNT;
        LU ← (stack) and (40000c); * check for Midas Present
        goto[MidasPause, alu#0];
        LPhi ← zero;
        LP ← CSMemStart;
        RTEMP1 ← zero; * Jump flag
        LoadPage[LRJPage];
        gotop[LRJenter];

loadmi:
        usectask;
        t←apc&apctask;
        rlink0←t;
        contemp←(3400c);        *last address + 1 of disk data
nextmi:
        t←contemp←(contemp)-1;
        pfetch1[Nova,initr2],call[LoadCSRet];
        t←contemp←(contemp)-1;
        pfetch1[Nova,initr0],call[LoadCSRet];
        t ← ldf[initr2,0,14];
        initr3←t;
        t←contemp←(contemp)-1;
        pfetch1[Nova,initr1],call[LoadCSRet];
        MemAddr ← t ← (MemAddr) + 1;* copy into memory
        pstore1[Nova,initr2],call[LoadCSRet];*write into memory
        MemAddr ← t ← (MemAddr) + 1;* copy into memory
        pstore1[Nova,initr0],call[LoadCSRet];*write into memory
        MemAddr ← t ← (MemAddr) + 1;* copy into memory
        pstore1[Nova,initr1],call[LoadCSRet];*write into memory
        lu←(initr3)xnor(170000c);       *look for m-i address = 7777
        goto[TransferDone,alu=0];
        t←ldf[contemp,7,10];    *look for mem address = 3001
        goto[nextmi,alu#0];
        t←(2000c);
        pfetch1[Nova,xCNT],call[LoadCSRet];
        nop;
        pstore1[DMA,xCNT,IOCBDiskAdr!],call[LoadCSRet];*next disk address
        apc&apctask←rlink0;
LoadCSRet:
        Return;

TransferDone:   t←(2000c);
        pfetch1[Nova,contemp], call[LoadCSRet]; *get next sector address
*       xCNT ← NextDiskAddr;
*       stkp←xCNT;
        T ← (contemp);
*       stack ← (T); * save the next disk address
        BootDiskAddr ← (T); * save the next disk address
        GOTO[BootLoader];

BootLoader:     LoadPage[InitPage];
                gotop[DiskBoot];

MidasPause:     NWW ← (100000c), ;
MidasWait:      BreakPoint, goto[MidasPause];


*************** added the following for overlay, will be over-written later

                OnPage[LRJPage];

Set[LRJBase,Add[lshift[LRJPage,10],300]];
```

```
LRJenter:
        t ← xfTemp ← 1c, AT[LRJBase, 0];
        nop, AT[LRJBase, 1];* wait for write of xfTemp to avoid bypass problem
LRJloop:
        pfetch1[LP,xBuf2],call[LRJIncCount], AT[LRJBase, 2];
        pfetch1[LP,xBuf],call[LRJIncCount], AT[LRJBase, 3];
        pfetch1[LP,xBuf1],call[LRJIncCount], AT[LRJBase, 4];
        T ← ldf[xBuf2,0,14], AT[LRJBase, 5];     * address
        xBuf3 ← T, AT[LRJBase, 6];
        lu ← (xBuf3) XNOR (170000c), at[LRJBase, 7]; *look for m-i address = 7777
        T ← xBuf2, goto[RamLoaded,alu=0], AT[LRJBase, 10];
        LU ← xBuf, AT[LRJBase, 13];
        APC&APCTASK ← xBuf3, AT[LRJBase, 11];
        WRITECS0&2, AT[LRJBase, 14];
        LU ← xBuf1, AT[LRJBase, 15];
        APC&APCTASK ← xBuf3, AT[LRJBase, 16];
        WRITECS1, AT[LRJBase, 17];
        T ← xfTemp, goto[LRJloop], AT[LRJBase, 20];

LRJIncCount:
        T ← xfTemp ← (xfTemp) + 1, goto[JumpRet], AT[LRJBase, 21];

RamLoaded:
        RTEMP1, GOTO[.+2,Rodd], AT[LRJBase, 12];          * odd if no jump
        APCTASK&APC ← (xBuf1), call[JumpRet], AT[LRJBase, 22];* set TPC for return
        T ← (GetRSpec[103]) xor (377c), AT[LRJBase, 23];
        RTEMP ← FFaultAdd, AT[LRJBase, 24];
        Stkp ← RTEMP, RTEMP ← T, AT[LRJBase, 25];
        Stack ← (Stack) or (1c), AT[LRJBase, 26];
*       loadpage[4], AT[LRJBase, 27];
*       Stkp ← RTEMP, gotop[P4Tail], AT[LRJBase, 30];
        nop, AT[LRJBase, 27];*********will be over-written later
        nop, goto[.], AT[LRJBase, 30];*********will be over-written later

JumpRet:
        RETURN, AT[LRJBase, 31];

*************** following are for linkage, will be over-written later

                  OnPage[opPage3];
Kfcr:   goto[.], at[KFCRLoc];

P7Tail: goto[.], at[P7TailLoc];

                  OnPage[xfPage1];
Loadgc: goto[.], at[Loadgcloc];

                  OnPage[nePage];
neNoskip:         goto[.], at[neNoskipLoc];

        END;
```

```
*  File RDC.mc
*  Last edit by Jim Frandeen October 14, 1979  9:12 PM
*  This version preloads 16 words for write operations.
*******************************************************************
Insert[RdcDefs];
TITLE[RigidDiskController];
SET TASK[RdcTask];              *currently 11b = 9d
ON PAGE[RdcPage];

RdSectorWakeup:
%Come here every time we get a sector wakeup.  We get a sector wakeup at the end of every data sector. 1383 sectors go by every second, s
**o we get a wakeup every 723 microseconds. When we wake up, look and see what we were doing when we went to sleep so we can pick up wher
**e we left off. We were doing one of the following:
Waiting for a new command (State Idle);
Waiting for a drive to come ready (State DriveChange);
Waiting for a seek to complete (State SeekWait);
Waiting for a sector to come around on the disk to execute a command (State SectorWait);
Waiting for a transfer operation to complete so we can check for errors. (State DataTransfer).%
*DiskStatus has just been read from the Controller. First, update CurrentSector so we know what sector is about to  go by. Then go do som
**ething depending on what State we are in.

LU←(RdcDiskStatus) AND (RdcSector0);    *Test for sector 0.
RdcCurrentSector←(RdcCurrentSector)+1,  *Update current disk sector.
        Skip[ALU=0];    *Skip if not sector 0.
RdcCurrentSector←0C;
Dispatch[RdcState, 15,3];       *Get State for dispatch
T←RdcCSBptr,    *Load displacement to fetch IOCBptr from CSB.
        Disp[.+1];      *Dispatch on state pointer
RdcSectorTimeOutCount ← (RdcSectorTimeOutCount)-1,
        GoTo[RdSectorWait], AT[RdcBase, RdcSectorWait!];
LU←(RdcDiskStatus) AND (RdcSeekComplete),       *Check seek complete
        GoTo[RdSeekWait], AT[RdcBase, RdcSeekWait!];
LU←(RdcDiskStatus) AND (RdcDevSelOK),   *Check device selected
        GoTo[RdDriveChange1], AT[RdcBase, RdcDriveChange!];
T←LSH[RdcState, 14],    *Prepare State for EndCommand.
        GoTo[RdEndTransfer], AT[RdcBase, RdcDataTransfer!];

RdIdle:
*Come here when we are in the Idle state.  We are just hanging around waiting for TheFace to chain a new IOCB onto the CSB. TheFace is ou
**r link to TheOutsideWorld. When TheOutsideWorld wants to execute a disk command, he calls TheFace. TheFace then constructs an Input Out
**put Control Block (IOCB) and chains it onto the Controller Status Block (CSB). When we wake up, we look to see if there is anything to
**do.
*When we arrive here, T points to the CSB. Load IOCBptr and Synch from the first two words of the CSB. Look to see if Synch is negative.
**If so, TheOutsideWorld wants us to quit processing IOCBs.
Pfetch2[RdcZeroBase, RdcIOCBptr],       *Fetch IOCBptr and Synch from CSB.
        AT[RdcBase, RdcIdle!];
LU←RdcSynch,    *Test Synch
        GoTo[RdIdle1,R>=0];

*Well, it's time for a break.  TheFace says to knock it off for a while and not process any more IOCBs until he gets things figured out.
**We could get out of here quicker if we didn't have to check Synch. But we must check it every time, even if there is nothing to do. The
**Face will not be satisfied unless we say "I Got it" by setting bit 0 of the controller word in the CSB.
T←(RdcCSBptr)+(RdcCSBcontroller);       *Set displacement to store Synch.
Pstore1[RdcZeroBase, RdcSynch]; *Store <0 in csb.controller.
        GoTo[RdClearWakeup];

RdIdle1:
*Come here if Synch is not negative. We might have work do. Test IOCBptr to see if it is non-zero.
T←RdcIOCBptr;   *T points to IOCB.
RdIdle2:
*Come here from Chain when a command has been processed, and see if there is another IOCB waiting.
RdcState←RdcIdle,
        GoTo[RdNewIOCB,ALU#0];  *If IOCB is waiting
NOP;            *Placement constraint
*Well, shoot, there is nothing for us to do this time. Oh well! Look at the Synch word and see if any Mesa processes need to be notified
**about this sector wakeup. If not, skip the call to DoInt to save time. We need to split as fast as possible and give the machine to som
**eone else.
RdEndSectorWakeup:

*Synch could be used to schedule Mesa processes that want to know about this sector wakeup.  Since Pilot is not currently using this feat
**ure, this has been is disabled to save space.  The following comments could be removed to enable the feature for Pilot. If DOLang would
** allow more than one ComChar, we could make this a comditional assembly!!
%*********Start of Pilot Sector Wakeup Code*********************
T←(RdcCSBptr)+(RdcCSBsynch);
Pfetch1[RdcZeroBase,RdcSectorSynch];    *Fetch Synch word from CSB.
LU←RdcSectorSynch;
GoTo[RdClearWakeup,ALU=0];      *If Synch is zero.
T←(RdcSectorSynch) OR (RdcAllowTask),
        Call[RdTaskIfNotZero];


LoadPage[0];
CallP[DoInt],   *Set NWW and IntPending; uses registers 0,1.
        IOStrobe;       *Terminate the wakeup
*Continue here after the next sector wakeup.
Input[RdcDiskStatus,RdcStatus], *Read status register from controller.
        GoTo[RdSectorWakeup];

%*********End of Pilot Sector Wakeup Code**********************

*!********Start of MicrocodeDriver Code************************
T←(RdcCSBptr)+(RdcCSBsynch);
Pfetch1[RdcZeroBase,RdcSectorSynch];    *Fetch Synch word from CSB.
RdcTemp←RdcWakeupReg;   *Load address of NWW.
T←(GetRspec[103]) XOR (377C);   *Save Stkp, right side up.
Stkp←RdcTemp,
        RdcTemp←T;      *Set Stkp to NWW, save old Stkp in Temp
T←RdcSectorSynch;       *Get bits to OR into NWW.
Stack←(Stack) OR (T);   *Set NWW.
Stkp←(RdcTemp), *Set DriveChange to zero
        Call[RdTaskIfNotZero];
```

```
*!********End of MicrocodeDriver Code**************************

RdClearWakeup:
*Terminate the wakeup and go to sleep until the next sector wakeup.
Call[RdStrobe];

*Continue here at the next sector wakeup.
Input[RdcDiskStatus,RdcStatus], *Read status register from controller.
        GoTo[RdSectorWakeup];

RdNewIOCB:
*Boy, Wow!! We have something to do! Time's a wast'n! When the Controller  turns on our wakeup latch, we are only 56 bytes from the secto
**r mark. We only have 60 microseconds after sector wakeup to get ready and send a command to the Controller. No telling how much time ha
**s already gone by. We didn't get our wakeup call until we were the highest priority task. And the evil DiskController runs at a higher
**priority task level. We have already executed 9 instructions since wakeup. If the header shows up before we send the command, we will g
**et IOAtten and ServiceLate. And you know what that means! We will have to wait one more disk revolution. And TheOutsideWorld will not b
**e pleased!
*A rule of the game is that we have to task to give up control of the CPU after every dozen or so microinstructions.  We must task twice
**before we can get our command ready, and once more before we send the header data to the Controller. And we don't know when we will get
** scheduled again after a task. The evil DisplayController is a glutten for CPU cylces. If we don't task often enough, the display will
**flicker. And the OutsideWorld will really be pissed! At the same time, the sector mark is less than 60 microseconds away. So we better
**hurry! Try to avoid branches that cause burps, and beat it down to State DataTransfer.
*T and IOCBptr point to the new IOCB. Fetch the new disk address from the first two words of the IOCB into NewDriveCylinder and NewHeadSe
**ctor. The next two instructions calculate a pointer to CSB.diskAddress[of the drive we are about to access].
Pfetch2[RdcZeroBase,RdcNewDriveCylinder];

T←(RdcCSBptr)+(RdcCSBdiskAddress),       *T points to CSB.diskAddress[drive 0].
        Call[RdTaskIfNotZero];

T←(LDF[RdcNewDriveCylinder,0,2])+T;      *T points to CSB.diskAddress[current drive].

*There was our first task switch after 12 instructions. Here we go again.
LU←(RdcDiskAddressPtr) XOR T;   *Test for drive change since last command.
RdcDiskAddressPtr←T,     *Update pointer to CSB.diskAddress[drive].
        Skip[ALU=0];     *If no drive change.
RdcState←RdcDriveChange;         *New drive this wakeup.
*Fetch the current cylinder address of the drive we are about to reference from the CSB. This will be set to -1 if we must recalibrate.
Pfetch1[RdcZeroBase,RdcCurrentCylinder];

*Send the drive and head to the controller. Drive is in the two low order bits of DiskAddressPtr. DiskAddressPtr points to CSB.diskAddres
**s[drive]. Head is in NewHeadSector.
T←(LDF[RdcNewHeadSector,0,10]); *Head is in bits 14-17
T←(LSH[RdcDiskAddressPtr,4]) OR T;       *Drive is in bits 12-13
RdcTemp←T;
Output[RdcTemp, RdcDrive/Head]; *Send drive and head to controller.

*See if the drive has changed since the last command. The State will be Idle (an even state) or DriveChange (an odd state).
LU←(RdcState),
*This is our first obstacle. If the drive has changed since the last command, we will have to wait for the Controller to tell us the new
**drive is ready.
        GoTo[RdDriveChange,R Odd];       *If drive change

RdDriveReady:
*The drive has not changed. See if we need to recalibrate.

LU←(RdcCurrentCylinder),
        Skip[R>=0];      *If recal not required.
*Well, this is obstacle number two.  The drive must be recalibrated before we can execute any commands. This means we must move the disk
**arm back to track 0. Seek errors cause this to happen. GoTo Recal and continue at TestForSeek when the drive has been recalibrated. If
**no seek is required after recalibration, continue at SeekComplete after the heads have settled.
LU←(RdcDiskStatus) AND (RdcTrack0),
        GoTo[RdRecal];

*The drive is ready, and we we do not need to recalibrate.  See if we need to seek. We will not need to seek if the disk address is for o
**ne of the  fixed heads or if CurrentCylinder is equal to NewCylinder. Test for the same cylinder first to save time.
T←RdcNewCylinder←(LDF[RdcNewDriveCylinder,10,10]);      *Remove head address from NewHeadCylinder.
RdTestForSeek:
LU←(RdcCurrentCylinder)-T;       *Test for different cylinder.
T←(RdcIOCBptr)+(RdcIOCBnext),   *Prepare to fetch.
        GoTo[RdSeekComplete,ALU=0];      *If CurrentCylinder = NewCylinder

*Here we are stuck at the last obstacle. It looks like we will have to seek before we can execute a command. Go move the disk arm to the
**new cylinder and continue at SeekComplete when the heads have settled at the new cylinder.
LU←(LDF[RdcNewHeadSector,4,1]), *Test for fixed heads.
        GoTo[RdSeek];

RdSeekComplete:
*We made it past the last obstacle. The disk arm is sitting at the right cylinder. Get the command ready to send to the Controller.
*Fetch the rest of the data we need from the next four words of the IOCB: NextIOCB points to the next IOCB in the chain, if any (Note: we
** must load this again at RdChain); Command is the command to execute; LongPointer and LongPointer1 point to the data for the transfer o
**peration. Note that this operation destroys registers that were being used for other things during the seek. If the command is seek onl
**y, we are through with this IOCB.
Pfetch4[RdcZeroBase, RdcNextIOCB];       *T=(RdcIOCBptr)+(RdcIOCBnext).

RdcCommand←(RdcCommand) AND NOT (RdcSeekBits),
        Call[RdTaskIfNotZero];
Output[RdcZeroBase, RdcMemBuffAdr];      *Set MemBufAdr to zero.

*Continue after our second task switch after 14 instructions.  This was the first reasonable place to task.
LU←LDF[RdcCommand,10,10];        *Header, label, and data commands
RdcCommand←(RdcCommand) OR (RdcAllowWake),
        GoTo[RdEndCommand,ALU=0];        *If seek only

*Continue if command is not seek only.

*~********Start of Pilot Code**********************************
*Get LongPointer ready to point to the data area in memory.  If BP[0:23] is a base pointer, BP[0:7] is in bits 0-7, and BP[0:7]+1 is in b
**its 8-15.
RdcLongPointer1←T←LSH[RdcLongPointer1,10];
RdcLongPointer1←(RSH[RdcLongPointer1,10])+1;
```

```
RdcLongPointer1←(LDF[RdcLongPointer1,10,10]) OR T;
*~*******End of Pilot Code*************************************

LU←(RdcCommand) AND (RdcDataWriteOrVerify);
T←RdcIOCBptr,   *Set displacement for Fetch.
        GoTo[RdPrepareSectorWait,ALU=0];        *If not write or verify.

*Continue if write or verify operation.  Preload the Controller buffer with the header and label data and 16D words of data.
IOFetch20[RdcZeroBase,RdcOutput],        *Send the header data.
        Call[RdTaskIfNotZero];
T←0C;           *Set displacement for IOFetch.
IOFetch20[RdcLongPointer,RdcOutput];    *Transfer 20 data words to  the Controller buffer.
RdPrepareSectorWait:
RdcSectorTimeOutCount←RdcSectorTimeOutWakeUps;
RdSectorWait:
*Here we are ready to send a command to the Controller.  Look to see if the sector coming up is the one we want to access. CurrentSector
**is what we think the next sector will be.

T←LDF[RdcNewHeadSector,10,10],   *T = sector specified by command.
        GoTo[RdSectorTimeOut, ALU=0];   *If sector time out
LU←(RdcCurrentSector) XOR T;
RdcState←RdcDataTransfer,
        GoTo[RdDataTransfer,ALU=0];     *Go for it!!

*Can you believe it? We hurried all the way down here, and this isn't even the sector we need to access. Go back to sleep until the next
**sector wakeup. Then continue at SectorWait above, and check that sector.
RdcState←RdcSectorWait,
        GoTo[RdEndSectorWakeup];        *Go back to sleep.

RdDataTransfer:
*Here we are at the right sector. Send the command to the Controller.
Output[RdcCommand, RdcDevOp];   *Send the Command.
*Hope we made it on time! We won't know until we try to read the header. If we get IOAtten here because we missed the sector, we will sti
**ll get the header wakeup, so we plunge ahead. Tune in at ReadHeader for the next exciting episode!

LU←(RdcCommand) AND (RdcDataWriteOrVerify);
T←RdcIOCBptr,   *Set displacement for Fetch.
        GoTo[RdReadHeader,ALU#0];       *If write or verify.

*Continue if not write or verify.  We have not yet loaded the header and label data into the Controller buffer.
Output[RdcZeroBase, RdcMemBuffAdr];     *Set MemBufAdr to zero.
NOP;            *Two instructions after Output before memory instruction.
LU←(RdcCommand) AND (RdcWriteHeader);   *Test for writing headers
IOFetch20[RdcZeroBase,RdcOutput],       *Send the header data.
        Skip[ALU=0];    *If not writing headers.
*If we are writing headers, we are done until the next sector wakeup. At that time, we will check to see if we got any errors. Terminate
**this wakeup and go to sleep.
GoTo[RdEndSectorWakeup];

GoTo[RdReadHeader];     *This extra instruction is for placement constraint.

RdReadHeader:
*We have done all we can for now. Terminate this wakeup and go to sleep until the header field on the disk shows up. The header field con
**tains the address of the next sector.
Call[RdStrobe]; *Terminate the wakeup.

*Continue here when the Controller has read the header field from the disk into its buffer. Transfer the header information from the Cont
**roller's buffer into the IOCB. This is the exciting moment, ladies and gentlemen! We will soon know  if we got our command ready in tim
**e.
T←0C,           *Set MemBuffAdr to zero.
        Call[RdPrimeIdata];     *Start the Controller.
T←RdcIOCBptr;   *Set base for IOStore
RdcTemp←RdcTemp;        *Interlock for PrimeIdata
IOStore4[RdcZeroBase, RdcInput],        *Read the header data into the first four words of the IOCB.
*Well, we blew it! We got IOAtten from the Controller, and that means trouble! Our command was probably too late. Sigh!
        GoTo[RdHeaderIOAtten, IOAtten];

*Well, we made it past the header. We win part one. But the game is not over. The bits are flying fast and furiously now. The SA4000 tran
**sfers seven million bits per second. Will we catch them all as the disk fly by? Or will the evil DisplayController steal our CPU cycles
** and cause ServiceLate or RateError? Tune in at EndTransfer for the exciting conclusion!!

LU←(RdcCommand) AND (RdcLabelReadOrVerify);     *For test at EndLabel
LU←(RdcCommand) AND (RdcDataWriteOrVerify),
        GoTo[RdEndLabel,ALU=0]; *If no label read or verify

*Come here to read or verify label.  Issue IOStrobe to cancel the Header field wakeup and go to sleep.
Call[RdStrobe]; *Terminate the wakeup.

*Continue at the first label field wakeup.  Read the first four label words into IOCB words 4 through 7.  Issue IOStrobe to terminate the
** wakeup and go to sleep.
T←6C,           *Set MemBuffAdr to 6.
        Call[RdPrimeIdata];     *Start the Controller
T←(RdcIOCBptr)+(RdcIOCBlabel);  *Set base for IOStore.
RdcTemp←RdcTemp;        *Interlock for PrimeIdata
IOStore4[RdcZeroBase,RdcInput], *Read the first four label words into IOCB locations 4 through 7.
        Call[RdStrobe]; *Terminate the wakeup.

*Continue after the second label field wakeup.  Read the second four label words into IOCB locations 10B through 13B.
T←(RdcIOCBptr)+(RdcIOCBlabel+4);
IOStore4[RdcZeroBase,RdcInput], *Read the next four words of the label into IOCB locations 10-13B.
        Skip[NoAtten];  *If not IOAtten

*Come here if we have IOAtten due to a label error. Set bit 0 in State to remember we got a label error so we can post LabelError in the
**status. We will continue to get data wakeups, so we plunge ahead.
RdcState←(RdcState) OR (RdcBit0);
LU←(RdcCommand) AND (RdcDataWriteOrVerify);

RdEndLabel:
*If write or verify data, GoTo WriteData.  If data read, GoTo ReadData; otherwise skip data entirely.
LU←(RdcCommand) AND (RdcReadData),      *Test for data read.
        GoTo[RdWriteData,ALU#0];        *If data write or verify.
```

```
GoTo[RdReadData,ALU#0]; *If read data
GoTo[RdClearWakeup];     *Skip data altogether.

RdWriteData:
*Come here to write or verify data. Issue IOStrobe to cancel the wakeup and go to sleep.
RdcTemp←40C,     *to set MemBuffAdr
        Call[RdStrobe]; *Terminate the wakeup.

*Continue after the first data field wakeup. Set MemBufAdr to the address following the last address loaded with data in the sector wake
**(20 header and label + 20 data = 40).
Output[RdcTemp, RdcMemBuffAdr];
*We must have two instructions after Output before a memory instruction.
RdcTemp←T←20C;  *Set displacement for IOFetch and Interlock.


RdWriteData2:
*Repeat the following loop 17B (15D) times: Transfer 20 (16D) words to the Controller buffer. Issue IOStrobe to terminate the wakeup. Go
**to sleep. Continue at the next data wakeup. Use SectorTimeCount as a temp to set to 370 so we can subtract T from 370 to test for the l
**ast transfer. This also makes another instruction after the Output and makes an even location to branch to on ALU>=0.
RdcLoopTest←370C;

*We execute IOFetch4 four times instead of one IOFetch20. Again due to the famous Redell Syndrome.
IOFetch4[RdcLongPointer,RdcOutput],       *Transfer 4 more data words to  the Controller buffer.
        Call[RdTplus4NoTask];   *Increment T and Temp by 4.
IOFetch4[RdcLongPointer,RdcOutput],       *Transfer 4 more data words to  the Controller buffer.
        Call[RdTplus4]; *Increment T and Temp by 4 and task.
IOFetch4[RdcLongPointer,RdcOutput],       *Transfer 4 more data words to  the Controller buffer.
        Call[RdTplus4NoTask];   *Increment T and Temp by 4.
IOFetch4[RdcLongPointer,RdcOutput],       *Transfer 4 more data words to  the Controller buffer.
        Call[RdStrobe]; *Terminate the wakeup.
*Continue here after the next data wakeup. T and Temp will be 374 the last time through the loop.
LU←(RdcLoopTest)-T;     *Test for last transfer
RdcTemp←T←(RdcTemp)+(4C),          *Increment displacement.
        GoTo[RdWriteData2,ALU>=0];         *If not last data transfer

*Continue here on the 16th wakeup. Ignore by issuing IOStrobe  and go to sleep. The next wakeup is equivalent to sector wakeup.
GoTo[RdClearWakeup];

RdReadData:
*Come here for data field read operation. Issue IOStrobe to terminate the header field or label field wakeup and go to sleep.
Call[RdStrobe]; *Terminate the wakeup.

*Continue here on the first data field wakeup.  Set MemBufAdr to 21B. Do PrimeIData to start the controller.
T←21C,          *Set MemBuffAdr to 21
        Call[RdPrimeIdata];
RdcTemp←T←0C;    *T is displacement for IOStore and Interlock.

RdReadData2:
*Repeat the following loop 16D times: Read 16D words into the data buffer. Issue IOStrobe to terminate the wakeup and go to sleep.
IOStore4[RdcLongPointer,RdcInput],         *Transfer 4 bytes from Controller buffer to memory.
        Call[RdTplus4NoTask];    *Increment T and Temp by 4.
IOStore4[RdcLongPointer,RdcInput],         *Transfer 4 bytes from Controller buffer to memory.
        Call[RdTplus4]; *Increment T and Temp by 4 and task.
IOStore4[RdcLongPointer,RdcInput],         *Transfer 4 bytes from Controller buffer to memory.
        Call[RdTplus4NoTask];   *Increment T and Temp by 4.
IOStore4[RdcLongPointer,RdcInput],         *Transfer last 4 bytes bytes from Controller buffer to memory.
        Call[RdStrobe]; *Terminate the wakeup.
LU←(RdcTemp) XOR (374C);          *Test for all bytes transferred.
RdcTemp←T←(RdcTemp)+(4C),         *Increment displacement.
        GoTo[RdReadData2,ALU#0];          *If not last transfer

*Come here after the 17th wakeup. When the last block has been transferred, go read status and do end of  transfer processing.
Input[RdcDiskStatus,RdcStatus], *Read status register from controller.
        GoTo[RdSectorWakeup];

RdEndTransfer:
*Well,  here we are boys and girls, at the wakeup after the data transfer.  And the winner is.......
*T=LSH[RdcState, 14].
RdcDiskStatus←(RdcDiskStatus) OR T,      *DiskStatus was loaded at Wakeup.
*Well, shit! We got all the way down here only to find we have an error.  This could be serious!
        GoTo[RdDataIOAtten,IOAtten];

*Wheww!!! We made it! Now go post the completion status in the IOCB so TheFace will know what happened. Again the mighty RigidDiskControl
**ler triumphs over the evil DisplayController!!!!!
*****************************************************************
*This is a temporary patch to see if the last command was a write, and if so to verify it.
%
LU←(RdcCommand) AND (4C);          *Test for write data
RdcCommand←(RdcCommand) AND NOT (7C),    *Turn off data field bits.
        GoTo[RdNotWrite,ALU=0]; *If not write command
RdcCommand←(RdcCommand) OR (1C);          *Turn on verify data bit.
RdcState←RdcSectorWait,
        GoTo[RdEndSectorWakeup];
RdNotWrite:
%
*****************************************************************
T←(RdcIOCBptr)+(RdcIOCBcompletion),       *Set displacement to post status.
        GoTo[RdEndCommand2];

RdEndCommand:
*Come here when the command has been completed. We are through with this IOCB. If the command was seek only, State is Seek; otherwise Sta
**te is DataTransfer or one of the error states. Post Status in IOCB.completion. (State is in bits [0..3]). Save the old IOCBptr in the C
**SB. NextIOCB points to the next IOCB in the chain, if any. If any type of error occurred, NextIOCB will be zero to stop processing IOCB
**s. Fetch IOCB.synch into CommandSynch. The transfer mask may be used to cause a Mesa process interrupt at the end of the command. Note
**that this is not the same as CSB.synch, which is still in Synch.
T←LSH[RdcState, 14];     *Status will have State in the high order 4 bits.
RdcDiskStatus←(RdcDiskStatus) OR T;      *DiskStatus was loaded at Wakeup.
T←(RdcIOCBptr)+(RdcIOCBcompletion);      *Set displacement
RdEndCommand2:
Pstore1[RdcZeroBase, RdcDiskStatus],     *Post status in IOCB
        Call[RdTaskIfNotZero];
```

```
T←(RdcCSBptr)+(RdcCSBoldIOCB);  *Set displacement
Pstore1[RdcZeroBase, RdcIOCBptr];       *Save old IOCBptr in CSB.
T←(RdcIOCBptr)+(RdcIOCBsynch);  *Set displacement to fetch IOCB.synch.
Pfetch1[RdcZeroBase, RdcCommandSynch];
*Now we call DoInt to schedule the Mesa processes that want to know about end of command. We don't test CommandSynch for zero since Pilot
** always wants to know about end of command.
*~********Start of Pilot Code*************************************
T←RdcCommandSynch,      *Bits to OR into NWW
        LoadPage[0];
CallP[DoInt];   *Set NWW and IntPending; uses registers 0,1; no TASK
*~********End of Pilot Code**************************************

*!********Start of MicrocodeDriver Code************************
RdcTemp←RdcWakeupReg;   *Load address of NWW.
T←(GetRspec[103]) XOR (377C);   *Save Stkp, right side up.
Stkp←RdcTemp,
        RdcTemp←T;      *Set Stkp to NWW, save old Stkp in Temp
T←RdcCommandSynch;      *Get bits to OR into NWW.
Stack←(Stack) OR (T);   *Set NWW.
Stkp←(RdcTemp);
*!********End of MicrocodeDriver Code**************************

RdChain:
*Fetch the pointer to the next IOCB from the current IOCB. We must fetch this pointer at the end of the transfer in order to give the Dri
**ver a chance to prepare the next IOCB. Then hurry back to see if there is another IOCB in the queue. Sometimes TheFace give us a whole
**list of IOCBs all at once. If the next IOCB is for the next sector on the disk, we have to be ready before the next sector mark comes a
**round.  It's back to the races, folks!
T←(RdcIOCBptr)+(RdcIOCBnext);   *Set base to fetch IOCBnext
LU←(RdcState)-(RdcDataError);   *Test for error
Pfetch1[RdcZeroBase,RdcIOCBptr],        *IOCBptr←IOCBnext
        Skip[ALU<0];    *If no error
RdcIOCBptr←zero;        *Zero IOCBptr to stop processing IOCBs.
T←RdcCSBptr;
Pstore1[RdcZeroBase,RdcIOCBptr];        *Store pointer to next IOCB in CSB.
T←RdcIOCBptr,
        GoTo[RdIdle2];  *Go process next IOCB.
```

```
RdDriveChange:
*Come here if the new command references a different disk drive than the previous command.  We have to wait until the Controller has a ne
**w drive ready.  When the drive is ready, go back and continue where we left off.

RdcSectorTimeOutCount←RdcDriveChangeTimeWakeUps,
        GoTo[RdEndSectorWakeup];          *Go back to sleep.

RdDriveChange1:
*Continue here at the next sector wakeup when State is DriveChange. LU = (RdcDiskStatus) AND (RdcDevSelOK). See if device is ready.
RdcSectorTimeOutCount ← (RdcSectorTimeOutCount)-1,
        Skip[ALU=0];      *If drive is not yet ready.
GoTo[RdDriveReady];       *DevSelOK=1

*Come here if the disk is not yet operational (DevSelOK=0). Check for timeout.
Skip[ALU=0];      *If not timeout
*Come here if the device is not yet operational and timeout has not occurred.
GoTo[RdEndSectorWakeup];

*Come here if disk has timed out. We can't hang around forever waiting for the disk. TheFace will wonder what happened to us.
RdcState←RdcDiskTimeOut,GoTo[RdEndCommand];


RdRecal:
*Come here if CurrentCylinder is -1 to indicate that we must recalibrate the disk. Send negative seek commands to the Controller until Tr
**ack0 appears in the Status word. Wait for each seek to complete before sending another Command; otherwise we might get carried away and
** send too many seek commands.  This causes the disk arm to bang against the stop, and it makes an awful noise.
*Status AND Track0 is on LU. Strip the head address from NewHeadCylinder. If we go back to TestForSeek, we will need NewCylinder in T.
T←RdcNewCylinder←(LDF[RdcNewDriveCylinder,10,10]),
        Skip[ALU#0];      *If we are at track 0.
*We are not at track 0 yet.  Send a negative seek command and wait for the seek to complete. Return to Recal above when it has completed.
**
T←1C,    *Seek one cyllinder.
        GoTo[RdSeekNeg];
*We are at track 0. We still have to seek to NewCylinder from track 0.
RdcCurrentCylinder←0C,
        GoTo[RdTestForSeek,ALU#0];        *If NewCylinder#0
*NewCylinder is 0.  We will not need to seek, but we must wait for the heads to settle after recalibrating.
RdcHeadSettleCount←RdcHeadSettlingTimeWakeUps,
        GoTo[RdHeadSettle];

RdSeek:
*Come here if CurrentCylinder is not the same as NewCylinder. We need to seek, unless the address is for one of the fixed heads. Head add
**ress is in bits [0..7] of NewHeadSector. Bits[0..3] are not used. Bit 4 will be nonzero if fixed heads.

T←RdcNewCylinder,         *LU←(LDF[RdcNewHeadSector,4,1]).
        Skip[ALU=0];      *If moving heads.
*Fall through here if fixed heads. Come here when the heads have settled after a seek.
RdSeek2:
T←(RdcIOCBptr)+(RdcIOCBnext),    *Prepare to fetch.
        GoTo[RdSeekComplete];

*Continue if the disk address is for the  moving heads.  We really do need to seek.  Well, there is no need to hurry any more.  We have a
** long wait. Send seek commands to the Controller one at a time. Each command moves the arm one cylinder in the negative or positive dir
**ection.
T←(RdcCurrentCylinder)-T;         *T=CurrentCylinder-NewCylinder
GoTo[RdSeekNeg,ALU>=0]; *If CurrentCylinder > NewCylinder.
*Continue if seek direction is positive.
RdcCommand←RdcSeek+D,     *Set for positive direction seek.
        GoTo[RdSeekLoop];
RdSeekNeg:
RdcCommand←RdcSeek-D;     *Set seek command for negative seek.
T←(zero)-T;       *Seek counts up instead of down.
RdSeekLoop:
*Send seek commands to Controller.  T = number of tracks to seek in negative form. This crazy machine can add one to T, but it can't subt
**ract one from T. So we count up instead of down.
Output[RdcCommand,RdcDevOp];      *Send seek command, direction, and allow wake to Controller.

*Now we need to delay at least one microsecond before issuing another Output command. If we send the commands too fast, the Controller ge
**ts confused. We delay by tasking. Initialize TimeOutCount. We better find our cylinder before it times out. Set up HeadSettlingCount. W
**e will start to count down when the seek is complete.

RdcState←RdcSeekWait,
        Call[RdTaskIfNotZero];
RdcHeadSettleCount←RdcHeadSettlingTimeWakeUps,
        Call[RdTaskIfNotZero];
T←(zero)+T+1;             *Decrement # tracks to seek.
RdcSectorTimeOutCount←RdcSeekTimeOutWakeUps,
        GoTo[RdSeekLoop,ALU#0]; *If we need to send another seek command.
*Turn off the seek bit in the command, but leave AllowWake and Direction bits on. otherwise the Controller gets confused.
RdcCommand←(RdcCommand) AND (RdcAllowWakeAndSeekDirection);
Output[RdcCommand, RdcDevOp],     *Send command to Controller.

        GoTo[RdEndSectorWakeup];          *Go back to sleep until the next sector wakeup.

RdSeekWait:
*Now we must wait for the seek to complete. Go to sleep and continue here at each sector wakeup.  Check the Status word at each wakeup un
**til SeekComplete appears.
RdcSectorTimeOutCount ← (RdcSectorTimeOutCount)-1,
        GoTo[RdSeekWait2,ALU#0];

*Continue if seek has not yet completed.  See if seek has timed out.
GoTo[RdSeekTimeOut,ALU=0];        *If seek timed out.

*Continue if seek did not time out and seek is not complete.
GoTo[RdEndSectorWakeup];

RdSeekWait2:
*Seek has completed. See if this was a one step seek for recalibrating.
LU←RdcCurrentCylinder,
        GoTo[RdHeadSettle,R>=0];          *If not recalibrating
```

```
*The seek has completed, and we are recalibrating. Go see if we are at track 0 yet.
LU←(RdcDiskStatus) AND (RdcTrack0),
	GoTo[RdRecal];

RdHeadSettle:
*Come here when the seek has completed. Now we must wait for the heads to settle at the new cylinder. All that seeking was a shock to the
**ir little sensors. Go to sleep and count sector wakeups until we think the heads have settled down.
T←RdcDiskAddressPtr;	*Set base to point to csb.diskAddress[drive].
*Update CSB.diskAddress[drive]. We do not update CurrentCylinder. Do this first because it gives us a chance to task.
Pstore1[RdcZeroBase,RdcNewCylinder],
	Call[RdTaskIfNotZero];
RdcHeadSettleCount←(RdcHeadSettleCount)-1;
Skip[ALU=0];	*If heads have settled.
*If we came here from Recal, the State has not yet been set.
RdcState←RdcSeekWait,
	GoTo[RdEndSectorWakeup];

*The seek has completed, and the heads have settled.
GoTo[RdSeek2];
```

```
RdStrobe:
*This subroutine terminates the wakeup and goes to sleep until the next sector wakeup. Note: This does not work if IOStrobe and RETURN ar
**e on the same statement. We must delay before the RETURN.
IOStrobe,
	GoTo[RdTask];

RdTaskIfNotZero:
************Begin Code to Lock Out Task Zero on Task Switch*********
*This task switch will switch to any task except task 0. It is desirable to lock out task zero in the critical timing paths in order to r
**educe the chances of a rate error or service late. To let in task zero, make the lines between asterisks comments and change PrimeIdata
** to delay before the GoTo[RdTask].
LU←APCTask;
UseCTask,
	Skip[ALU=0];
NOP;
************End of Code to Lock Out Task Zero on Task Switch********
RdTask:
RETURN;

RdTplus4:
*Add 4 to Temp and T. This is used often by IOFetch4 and IOStore4. The first instruction must allow for the bypass of T. Since the last i
**nstruction was a memory instruction, T has not yet been updated.
NOP;
T←RdcTemp←(RdcTemp)+(4C),
	RETURN;

RdTplus4NoTask:
*This is just like RdTplus4, but it does not allow a task switch.
UseCtask;
T←RdcTemp←(RdcTemp)+(4C),
	RETURN;

RdPrimeIdata:
*This subroutine sets MemBuffAdr to the value in T and primes the Controller by Output to PrimeIdata. The instruction following the Call[
**RdPrimeIdata] must do an interlock of Temp to insure that the Output has been completed. We do not do the interlock before the return s
**o as not to hold up the CPU unnecessarily.
RdcTemp←T;	*T contains the value to write to MemBuffAdr.
Output[RdcTemp,RdcMemBuffAdr];	*Set MemBuffAdr.
Output[RdcTemp,RdcPrimeIdata],	*Start the Controller.
*We must delay before tasking after an Output. Task IfNotZero has d delay.
	GoTo[RdTaskIfNotZero];	*Return
```

```
*ERRORS COME THROUGH HERE.

RdHeaderIOAtten:
*Come here if we got IOAtten in state DataTransfer just after reading the Header. If the error was due to ServiceLate, we will retry if t
**he retry error counter has not yet gone negative; otherwise we report the error.
RdcState←RdcHeaderError;          *Set error code

RdTestError:
*State is HeaderError, DataError, or LabelError.
Input[RdcDiskStatus,RdcStatus], *Get Status from Controller.
        Call[RdTask];
Output[RdcTemp,RdcErrorReset];
LU←(RdcDiskStatus) AND (RdcErrorBitsLow);
LU←(RdcDiskStatus) AND (RdcServiceLate),
        GoTo[RdReportError,ALU#0];       *If BufErr, RdErr, WriteFault, or Ofault

*Continue if not BufErr, RdErr, WriteFault, or Ofault. Test for ServiceLate.
LU←(RdcDiskStatus) AND (RdcRateError),
        GoTo[RdServiceLate,ALU#0];       *If ServiceLate

*Continue if not ServiceLate. Test for RateError.
T←(RdcIOCBptr)+(RdcIOCBsyndrome)+1,      *Set up to fetch RateError retry count.
        GoTo[RdTestRetry,ALU#0];         *If RateError.

RdForceRecal:
*Continue here if no error bits are set in the Status word and we got IOAtten in the header field. We should not ever get header errors.
**Either we lost our place on the disk or there is garbage in the header. Force a recal  the next time we try a command.
*Come here from SeekTimeOut or SectorTimeOut.
T←RdcDiskAddressPtr;
RdcTemp←(zero)-1;         *Store zero in disk address
Pstore1[RdcZeroBase,RdcTemp],
        GoTo[RdEndCommand];

RdServiceLate:
*!********Start of MicrocodeDriver Code*************************
RdcSectorTimeOutCount←RdcSectorTimeOutWakeUps;
RdcState←RdcSectorWait,
        GoTo[RdEndSectorWakeup];         *Retry all ServiceLates.
*!********End of MicrocodeDriver Code***************************
T←(RdcIOCBptr)+(RdcIOCBsyndrome),        *Set up to fetch ServiceLate retry count.
        GoTo[RdTestRetry];
RdTestRetry:
*Come here if ServiceLate or RateError. Fetch the retry count from the Syndrome. Retry count+1 times. This means if the count is zero, we
** will retry once. Decrement this count, update it in the IOCB, and test for negative.  If negative, we give up; let TheFace figure out
**what to do. If the retry counter is not negative, we go back to State SectorWait and retry when the sector comes around again.
Pfetch1[RdcZeroBase,RdcTemp];    *Fetch retry count.
RdcTemp←(RdcTemp)-1,     *Decrement retry count.
        GoTo[RdRetry,R>=0];
*Come here if retry count has been exceeded.
RdErrorCountExceeded:    *This is a breakpoint label.
*Report error if error count has been exceeded.
GoTo[RdReportError];

RdRetry:
*Retry the command the next time the sector comes around.
Pstore1[RdcZeroBase,RdcTemp];    *Update retry count.
RdcState←RdcSectorWait,
        GoTo[RdEndSectorWakeup];         *Retry

RdDataIOAtten:
*IOAtten in state RdcData, after the data transfer. If the error is due to a RateError, we will retry if the error count in the IOCB is n
**ot yet negative; otherwise we will report the error. If the State is negative, the error occurred in the label field; otherwise the err
**or occured in the data field.
LU←RdcState,     *Test for label error.
        Skip[R>=0];      *If not label error.
RdcState←RdcLabelError, *Error occurrred in the label field.
        GoTo[RdTestError];
RdcState←RdcDataError,  *Error occurred in the data field.
        GoTo[RdTestError];

RdReportError:
T←23C,          *The Controller places the syndrome in words 23b and 24b.
        Call[RdPrimeIdata];      *Start the Controller.
RdcTemp←RdcTemp;         *Interlock for PrimeIdata
Input[RdcSyndrome,RdcInputBuffer];       *Get syndrome from Controller.
Input[RdcSyndrome1,RdcInputBuffer];      *Get next word of syndrome.
T←(RdcIOCBptr) + (RdcIOCBsyndrome);      *Set displacement to store syndrome.
Pstore2[RdcZeroBase,RdcSyndrome],        *Store syndrome in IOCB
        GoTo[RdEndCommand];

RdSeekTimeOut:
*From state Seek.  A seek time out error has occurred.
RdcState←RdcSeekTimeOut,GoTo[RdForceRecal];

RdSectorTimeOut:
*From state SectorWait.  The sector has timed out.
RdcState←RdcSectorTimeOut,GoTo[RdForceRecal];
```

```
ON PAGE[RdcInitPage];
*This is the RDC initialization code.  This is throw-away code that lives in a separate page from the RDC microcode.

*Redefine temporary registers for initialization.
RV[RdcReg0, ADD[RdcRegBase,0]];
RV[RdcReg1, ADD[RdcRegBase,1]];
RV[RdcReg2, ADD[RdcRegBase,2]];
RV[RdcReg3, ADD[RdcRegBase,3]];

RdcInit:
RdcReg0←0C, AT[RdcInitLoc];
RdcReg1←0C;
RdcReg2←0C;
RdcReg3←0C;
RdcZeroBase←0C;
RdcCSBptr←AND@[RDCCSBValue, 377]C;      *Set Controller Status Block address to 720.
RdcCSBptr←T←(RdcCSBptr) OR (AND@[RDCCSBValue, 177400]C);
Pstore4[RdcZeroBase, RdcReg0];  *Store zero into first four words of CSB.
RdcReg0←(zero)-1;
RdcReg1←(zero)-1;
RdcReg2←(zero)-1;
RdcReg3←(zero)-1;
T←(RdcCSBptr)+(RdcCSBdiskAddress);      *T points to disk addresses in CSB.
Pstore4[RdcZeroBase,RdcReg0];   *Set all disks to recalibrate.
RdcDiskAddressPtr←T;    *Set RdcDiskAddressPtr to point to CSB.diskAddress[drive 0].
*Set CurrentSector to a value higher than any sector. This will cause us to start counting at sector 0.
RdcCurrentSector←1000C;
Output[RdcReg0, RdcGeneralReset];       *Reset the Controller.
Output[RdcZeroBase, RdcDrive/Head];     *Set Controller disk and head to zero.
RdcCommand←RdcAllowWake;
Output[RdcCommand, RdcDevOp];   *Send Allow Wake to Controller.
*!********Start of MicrocodeDriver Code************************
*Microcode driver version
Call[RdcReturn];
*!********End of MicrocodeDriver Code**************************

LoadPage[RdcPage];
RdcState←RdcIdle,
        GoTo[RdClearWakeup];

*!********Start of MicrocodeDriver Code************************
RdcReturn:
RdcTemp←RdcDiagTask.1;
RdcTemp←(RdcTemp) OR (RdcDisk.2);
APCTask&APC←RdcTemp;
RETURN;
*!********End of MicrocodeDriver Code**************************
```

```
%
LOG

The first major revision to this code was to change all IOFetch16s to IOFetch4s. The IOFetch16 was causing a word to be duplicated on the
** disk and pushing every other word in the page down one word, pushing the last word on the page off the end. (This is the famous Redell'
** Syndrome). The Controller must have a new word ready for the disk every 2.2 microseconds. During the IOFetch, the Controller's buffer
**is locked out, and a new word cannot be transferred to the holding register. If enough branch burps cause the IOFetch16 to exceed the 2
**.2 microsecond time limit, a new word cannot be loaded into the holding register, and the previous word is duplicated.

October 14, 1979  4:34 PM: The new RDC bug has the following symptoms: Four word blocks of data were not getting preloaded into the Contr
**oller's buffer. On all write operations, 32 words of data were being loaded into the Controller's buffer after the Command had been sen
**t to the Controller. Sometimes 8, 12, or 20 words of this preload data would not get loaded into the buffer, and the data from the prev
**ious write operation would get written to the disk page. The Controller did not report any error.

Chuck Thacker suggested the following solution to the problem: Preload the buffer before sending the command to the Controller. I observe
**d some interesting "features" about the Controller when I tried this:

1. It is not a good idea to preload the buffer with header and label data if the command is not write or verify. If the command is a read
** operation, this causes ServiceLate to occur about 20 percent of the time. Because of this, we only preload the buffer for write or ver
**ify operations.

2. If we preload the buffer with 32 words of data, we get spurious IOAtten from the Controller in the header field. No error bits are set
** in the Status word, and the header data is correct (i.e., it is not a verification error). If we only preload 16 words, we do not get
**this spurious IOAtten.

3. If the command is verify label, read data, we must set MemBuffAdr to zero right after sending the command to the Controller; Otherwise
** we get spurious IOAtten in the label field, i.e., no error bits are set and there is no verify error.

4. If we preload the buffer for a write operation, we must not set MemBuffAdr to zero after we execute the command. This causes IOAtten i
**n the header field.  In fact, this may be the cause for the whole problem we were having.  Once in a while, maybe MemBuffAdr was gettin
**g set to zero before all of the data had been sent to the Controller.
%
```

```
*  File RDCDefs.mc
*  Last edit by Jim Frandeen September 19, 1979  2:19 PM
****************************************************************
%
This microcode is written to run under Pilot or under Super, the microcode driver written by Jim Katsiroumbas. The mode depends on two se
**ts of statements as follows:

To run under Pilot:

Insert[D0Lang];
NoMidasInit; LangVersion; MultDib;
Insert[GlobalDefs];
Set[PilotMode,1];

To run under Super:

Insert[SUPERDEFS];
NoMidasInit; LangVersion;
Set[PilotMode,0];

%
****************************************************************

Insert[D0Lang];
NoMidasInit; LangVersion; MultDib;
Insert[GlobalDefs];
Set[PilotMode,1];


IFE@[PilotMode,1,ComChar@[!],ComChar@[~]];

*!********Start of MicrocodeDriver Code************************
Set[RdcInitPage,RDCRPAGE1]; *Throwaway init code for SA4000
Set[RdcInitLoc,Add[Lshift[RdcInitPage,10],0]];
MC[RdcWakeupReg,10];
*!********End of MicrocodeDriver Code**************************

SET TASK[RdcTask];                  *currently 11b = 9d

SET[RdcBase, LSHIFT[RdcPage, 10]];

*The following are definitions for the registers that belong to the RDC microprogram task.

SET[RdcFirstRegister, LSHIFT[RdcTask,4]];       *The first register in the block assigned to the RDC task (currently 220B).
SET[RdcRegBase,AND@[60,RdcFirstRegister]];      *This makes all of the register addresses 6 bits long.
*Register 0 is destroyed across sector wakeups. In the DataError routine, it is used to fetch the syndrome from the Controller. SectorSyn
**ch is used to load CSB.synch at the end of sector wakeup.
RV[RdcSyndrome, ADD[RdcRegBase, 0]];
RV[RdcSectorSynch, ADD[RdcRegBase, 0]];

*Register 1 is destroyed across sector wakeups. Temp is used for a temporary register. In the DataError routine, it is used to fetch the
**syndrome from the Controller.
RV[RdcTemp, ADD[RdcRegBase,1]];
RV[RdcSyndrome1, ADD[RdcRegBase, 1]];

*DiskAddressPtr points to CSB.diskAddress[current drive]. It is used to update CSB.diskAddress when the drive changes and also to see if
**the drive has changed since the last command.
RV[RdcDiskAddressPtr, ADD[RdcRegBase,2]];

*CurrentSector contains the current sector number of the current drive selected. It is updated at each sector wakeup.
RV[RdcCurrentSector, ADD[RdcRegBase, 3]];

*The next four registers must be consecutive and quad word aligned: NextIOCB, Command, LongPointer, and LongPointer1. They are loaded tog
**ether from the IOCB when the state changes to DataTransfer.

*NextIOCB points to the next IOCB in the chain. It is not loaded from the IOCB until the DataTransfer state.
RV[RdcNextIOCB, ADD[RdcRegBase, 4]];

*Command is the command loaded from the IOCB. It is not loaded until state DataTransfer. Until then, it is used to send seek commands.
**
RV[RdcCommand, ADD[RdcRegBase, 5]];

*CurrentCylinder contains the current cylinder number of the current drive selected.  It is used to determine whether we need to seek, an
**d if so in what direction. It is loaded from the CSB for each command, and it is stored in the CSB after each seek. CSB.diskAddress[dis
**k index 0..3] contains the arm position of each drive. If -1, it means we must recalibrate. It is not used after the seek. LongPointer
**points to the data specified in the IOCB. It is not loaded from the IOCB until state DataTransfer.
RV[RdcLongPointer, ADD[RdcRegBase, 6]];
RV[RdcCurrentCylinder, ADD[RdcRegBase, 6]];

*HeadSettleCount is used to count sectors until the heads settle after a seek. LongPointer1 is the second word of LongPointer. It is not
**loaded from the IOCB until state DataTransfer.
RV[RdcHeadSettleCount, ADD[RdcRegBase, 7]];
RV[RdcLongPointer1, ADD[RdcRegBase, 7]];

*The next two registers must be consecutive and doubleword aligned: IOCBptr and Synch. They are loaded together from the IOCB.

*IOCBptr points to the IOCB of the current command..
RV[RdcIOCBptr, ADD[RdcRegBase, 10]];

*Synch is an interrupt mask loaded from the CSB. -1 means stop the Controller; otherwise it contains an interrupt mask used to schedule M
**esa processes after each sector wakeup. It is loaded at the beginning of the Idle State in order to see if it is necessary to stop proc
**essing IOCBs. SectorTimeOutCount is used to count sectors until timeout for states DriveChange, SeekWait, and SectorWait. CommandSynch
**is used to load the Synch word from the IOCB to schedule Mesa processes at the end of the command. (Note: this is not the same as CSB.s
**ynch). LoopTest is used in the DataTransfer State as a temporary loop control.
RV[RdcSynch, ADD[RdcRegBase, 11]];
RV[RdcSectorTimeOutCount, ADD[RdcRegBase, 11]];
RV[RdcCommandSynch, ADD[RdcRegBase,11]];
RV[RdcLoopTest, ADD[RdcRegBase,11]];

*CSBptr points to the Controller Status Block.
RV[RdcCSBptr, ADD[RdcRegBase, 12]];
```

```
*State tells what state we are in when we get a sector wakeup: Idle, SeekWait, SectorWait, DataTransfer. State is defined in more detail
**below. Bit 0 will be set after a data transfer if we got IOAtten in the label field.
RV[RdcState, ADD[RdcRegBase, 13]];

*The following two registers must be together and doubleword aligned: NewDriveCylinder and NewHeadSector. They are loaded together from t
**he IOCB.

*NewDriveCylinder contains the drive and cylinder address for the current command, loaded from the IOCB. The Drive bits are stripped off
**during the test for seek, and it becomes NewCylinder. It is only used to determine whether a seek is required. RegT is used to save the
** value of T for IOFetch4 and IOStore4 during data transfer operations. It is only used during the DataTransfer State.
RV[RdcNewDriveCylinder, ADD[RdcRegBase, 14]];
RV[RdcNewCylinder, ADD[RdcRegBase,14]];
RV[RdcRegT, ADD[RdcRegBase,14]];

*NewHeadSector contains the head and sector address of the current command, loaded from the IOCB. It is used to check and see if we are a
**t the right sector on the disk (if the Sector bits match CurrentSector) before starting a command. It is not used after we start the da
**ta transfer operation.
RV[RdcNewHeadSector, ADD[RdcRegBase,15]];

*DiskStatus contains the state of the Controller. It is loaded at each sector wakeup and when an error is detected.
RV[RdcDiskStatus, ADD[RdcRegBase, 16]];

*ZeroBase is used for memory commands to fetch and store data in the CSB and the IOCB. Since these control blocks are always in low memor
**y, T is used as the address. Since this is an odd base register, ZeroBase OR 1 is the same register.
Rv[RdcZeroBase,OR@[RdcRegBase,17]];


*Constants
MC[RdcAllowTask,100000];        *To allow task when returning from DoInt.
MC[RdcBit0,100000];

* The following definitions are for the Controller registers:
SET[RdcAddr, LSHIFT[RdcTask,4]];        *The device address of the RDC (currently 220B).
SET[RdcGeneralReset, 0];        *General Reset
SET[RdcStatus, 1];      *Controller and drive status
SET[RdcInputBuffer, 17];        *This register presents Buffer[RdcMemBuffAdr].
SET[RdcInput, ADD[RdcAddr,17]]; *Complete device address for above.
SET[RdcDrive/Head, 1];  *Drive is in bits 12-13 (10-11D), Head is in bits 14-17 (12-15D)
SET[RdcErrorReset, 2];  *Error Reset (data is ignored)
SET[RdcDevOp, 3];       *Disk commands and allow wake bit
SET[RdcOutput, ADD[RdcAddr,4]]; *Data loaded into this register is written into Buffer[RdcMemBufAddr].
SET[RdcMemBuffAdr, 6];  *This register holds the current pointer into the RDC's data buffer.
SET[RdcPrimeIData, 7];  *This register must be accessed by OUTPUT before reading the first word of each sector from the RdcBuffer.

*The following definitions are for the DevOpReg:
MC[RdcAllowWake, 4000]; *Allow wake bit must be set at each Output if wakeups are to be enabled.
MC[RdcAllowWakeAndSeekDirection, 5000];
MC[RdcSeekBits, 3000];  *Seek command bits 5-6
MC[RdcReadWriteBits, 377];      *Read-write bits 10-17
MC[RdcSeek-D, 6000];    *Seek command in negative direction (outward toward lower track numbers) plus allow wake.
MC[RdcSeek+D, 7000];    *Seek command in positive direction (inward toward higher track numbers) plus allow wake.
MC[RdcWriteHeader, 200];        *Write header
MC[RdcDataWriteOrVerify, 5];
MC[RdcLabelReadOrVerify, 30];
MC[RdcReadData, 2];

*The following definitions are for the RdcStatus word:
MC[RdcDevSelOK, 20];    *Device is selected and ready.
MC[RdcSeekComplete, 40];
MC[RdcTrack0, 100];
MC[RdcServiceLate, 2000];
MC[RdcRateError, 400];
MC[RdcSector0, 200];    *Physical sector 0
MC[RdcErrorBitsHigh, 400];      *RateError
MC[RdcErrorBitsLow, 17];        *BufErr, RdErr, WriteFault, Ofault

*The following definitions are for offsets in the IOCB:
MC[RdcIOCBlabel, 4];    *label: Label (words 4-13B)
MC[RdcIOCBlabel+4, 10]; *label: Label (words 10-13B)
MC[RdcIOCBnext, 14];    *next: IOCBLink
MC[RdcIOCBcommand, 15]; *Command
MC[RdcIOCBdata, 16];    *data: LONG POINTER
MC[RdcIOCBcompletion, 20];      *completion: Status
MC[RdcIOCBsynch, 21];   *synch: process wakeup mask
MC[RdcIOCBsyndrome, 22];

*The following definitions are for offsets in the CSB (Controller Status Block):
MC[RdcCSBsynch, 1];     *synch gets loaded into RdcState.  If bit 0 = 1, Quiet procedure says turn off Controller.
MC[RdcCSBdrive, 2];     *drive: Status.
MC[RdcCSBcontroller, 3];        *controller state.
MC[RdcCSBdiskAddress, 4];       *diskAddress: ARRAY[0..3] contains cylinder address only.  -1 is for recalibrate.
MC[RdcCSBoldIOCB, 17];  *oldIOCB: IOCBLink.

*Thefollowing memory locations are for debugging with the microcode driver.
MC[RdcRateErrorCount, 75];      *For debugging with microcode driver
MC[RdcServiceLateCount, 76];    *For debugging with microcode driver
MC[RdcHeaderErrorCount, 77];    *For debugging with microcode driver


*The following are timing definitions that are loaded into SectorTimeOutCount:
MC[RdcHeadSettlingTimeWakeUps, 37];     *Number of sector wakeups for head to settle after seek complete (20 miliseconds, or one disk rev
**olution).
MC[RdcSeekTimeOutWakeUps, 1000];        *Number of sector wakeups to wait until seek time out.
MC[RdcDriveChangeTimeWakeUps, 2];       *Number of sector wakeups to wait after drive change.
MC[RdcSectorTimeOutWakeUps, 100];       *Number of sector wakeups to wait until sector time out when in state SectorWait.
```

```
*The following definitions are for the controller State:
MC[RdcSectorWait, 2];
*Waiting for the sector on the disk (CurrentSector) to match the sector specified by the current command.
MC[RdcSeekWait, 3];
*The Controller is seeking the cylinder specified by the current command, and we are waiting for SeekComplete to appear in the Status wor
**d.
MC[RdcIdle, 4];
*This is the initial State.  The Controller is not executing a command. We are waiting for a new IOCB to be chained onto the CSB. (Note:
**we count on this state being even).
MC[RdcDriveChange, 5];
*The Controller enters this state to wait for a drive to come ready when the drive changes between commands. (Note: we count on this stat
**e being odd).
MC[RdcDataTransfer, 6];
*Data transfer is in progress. At the next sector wakeup, we will check IOAtten.

*The rest of the states are error states.
MC[RdcDataError, 12];
*Enter this state from state DataTransfer if any error is detected. Bits in the Status word indicate the problem.
MC[RdcSectorTimeOut, 13];
*Enter this state if the timeout occurs before the sector is found.
MC[RdcHeaderError, 14];
*Enter this state if IOAtten occurs in the Header field.
MC[RdcSeekTimeOut, 15];
*Enter this state from state Seek if a time out occurs while doing a seek.
MC[RdcDiskTimeOut, 16];
*Enter this state from DriveChange if time out occurs before the drive becomes ready.
MC[RdcLabelError, 17];
*Enter this state from state DataTransfer if IOAtten occurred while reading the label field.
```

```
insert[d0lang];

NOMIDASINIT;LANGVERSION;

        title[RS232Async];
*       Last modified by BRD on August 28, 1979  12:48 PM fixed GlobalDefs conflicts
*       modified by BRD on June 25, 1979  11:44 AM
*       Version 3.0

insert[GlobalDefs];
insert[RS232Defs];

* This microcode uses a timer to simulate a hardware wakeup for the frame task.  It uses
* Task 4 for the frame task, and task 5 for the bit task.


        SET TASK [RBTask];
        ONPAGE [RBPage];

* Wakeup occurred on input timer.  Get input bit value and dispatch on input state,

RIWake: $RIBitDisp[RState], AT [RIWakeLoc];
        DISP[RICheckStart], T ← (RS232) AND (RI232Data);

* Start bit detected, check bit to be sure its a start bit and not noise.

RICheckStart: GOTO [RIGood, ALU=0], T ← (RDataMask) AND (RDataSizeMask), AT [RIBitLoc,0];
RISetStart: LOADTIMER[RIHalfBitLo];
        LU ← (RDataMask) AND (RDataSlow);
        SKIPON [ALU#0];
        RETURN;
        NOP;
        NOP;
        GOTO [RTimerReturn], LOADTIMER[RIHalfBitHi];

* Good start bit, increment to next state and set whole bit timer

RIGood: RIData ← T;
RINextState: RState ← (RState) + (RIBStateIncr);
RISetTimer: ADDTOTIMER[RIFullBitLo];
        LU ← (RDataMask) AND (RDataSlow);
        SKIPON [ALU#0];
        RETURN;
        NOP;
        NOP;
        GOTO [RTimerReturn], LOADTIMER[RIFullBitHi];

* Data bit in.  Shift into partially assembled character.  If end of character, increment
* state to check stop bit.

RIShiftBit: RState ← (RState) XOR (T),                          AT [RIBitLoc,1];
        DBLGOTO [RINextState, RISetTimer, R ODD], RIData ← (RSH[RIData,1]) OR (T);

* Justify character and check parity bit.

        $RIJustDisp[RDataMask],                                 AT [RIBitLoc,2];
        DISP [RIJustify], RState ← (RState) XOR (T);
RIJustify:      GOTO [RIParity], RIData ← RSH[RIData,3],                AT [RIJustLoc,0];
        GOTO [RIParity], RIData ← RSH[RIData,2],                AT [RIJustLoc,1];
        GOTO [RIParity], RIData ← RSH[RIData,1],                AT [RIJustLoc,2];
RIParity:       $RParityDisp[RDataMask],                                AT [RIJustLoc,3];
        DISP [RIParNone], RState ← (RState) + (RIBStateIncr);
RIParNone:      GOTO [RICheckStop], RBTemp1 ← T,                        AT [RIParLoc,0];
RIParOdd:       GOTO [RIParChk], LU ← LDF[RState, RIParityBit, 1],      AT [RIParLoc,1];
RIParEven:      GOTO [RIParChk], LU ← (LDF[RState, RIParityBit, 1]) - 1,  AT [RIParLoc,2];
RIParOne:       GOTO [RIParChk], LU ← LDF[RBTemp1, RIParityBit, 1],     AT [RIParLoc,3];
RIParZero:      GOTO [RIParChk], LU ← (LDF[RBTemp1, RIparityBit, 1]) - 1,  AT [RIParLoc,4];
RIparChk:       GOTO [RIParGood, ALU#0];
        RIData ← (RIData) OR (RIDataParityErr);
RIParGood:      GOTO [RISetTimer];

* Stop bit in.  Check to verify that this is a stop bit (Data=1).  If not it's
* either framing error or a break.  Set break/framing error counter.

RICheckStop:GOTO[RITurnon,ALU#0],                               AT [RIBitLoc,3];
        GOTO [RINextState], RIData ← (RIData) OR (3000C);

* Checking for either framing error or break character.  We do that by decrementing the
* break/framing error counter each bit time until the line returns to the idle state.
* If the counter goes negative, we call it a break.  If not, it is just a framing error.

        SKIPON [ALU=0], RIData ← (RIData) - (400C),             AT [RIBitLoc,4];
        DBLGOTO [RINotifyBreak, RINotifyFrame, R<0], RIData ← RHMASK[RIData];
        SKIPON [R>=0], LU ← RIData;
        RIData ← (RIData) + (400C);
        GOTO [RISetTimer];
RINotifyFrame: GOTO [RITurnOn], RIData ← (RIData) OR (RIDataFrameErr);
RINotifyBreak: GOTO [RITurnOn], RIData ← (RIData) OR (RIDataBreak);

* Notify frame task that input has work

RITurnOn:       SKIPON [R<0], RState ← (RState) OR (OR@[RIActive!, RXActive!]C);
        LOADTIMER[RFrameTimer1];
        RState ← (RState) AND NOT (RIBStateMask);
        T ← (RPtrCSB) + (RIndexCharIn);
        PStore1[RBasePage0, RIData];
        GOTO [RISetStart];
```

```
        SET TASK[RBTask];
        ONPAGE [RBPage];

* Timer wakeup occurred on output.

ROWake: RStackSave ← pRSImage, AT [ROWakeLoc];
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        $ROBitDisp[RState];
        DISP[ROIdle], T ← Stack ← (Stack) AND (376C);

* Idle output state.  Used for idling line and sending stop bits.

ROIdle: GOTO [ROSendBit], T ← Stack ← (Stack) OR (1C),            AT [ROBitLoc,0];

* Get next input character.  Notify frame code.
* If we have a character, send start bit.  If no character to be sent, idle line.

ROStart:        SKIPON [R<0], RState ← (RState) OR (OR@[ROActive!,RXActive!]C),AT [ROBitLoc,1];
        LOADTIMER[RFrameTimer1];
        T ← (RPtrCSB) + (RIndexCharOut);
        PFetch1[RBasePage0, ROData];
        T ← (RDataMask) AND (RDataSizeMask);
        SKIPON [R>=0], ROData ← (LSH[ROData,10]) OR (T);
        GOTO [ROIdle], RState ← (RState) - (ROBStateIncr);
        T ← Stack;
        GOTO [RONextState], RS232 ← T;

* Send data bit

ROSendChar: T ← LDF[ROData,7,1],                                   AT [ROBitLoc,2];
        RState ← (RState) XOR (T);
        T ← Stack ← (Stack) OR (T);
ROSendBit:      RS232 ← T;
        DBLGOTO [RONextState, ROSetTimer, R ODD], ROData ← RSH[ROData,1];
RONextState: RState ← (RState) + (ROBStateIncr);
ROSetTimer: STKP ← RStackSave;
        ADDTOTIMER[ROFullBitLo];
ROLoadTimer: LU ← (RDataMask) AND (RDataSlow);
        SKIPON [ALU#0];
        RETURN;
        NOP;
        NOP;
        GOTO [RTimerReturn], LOADTIMER[ROFullBitHi];
ROSetTimer1: GOTO [ROLoadTimer], LOADTIMER[ROFullBitLo];

* Send parity bit if parity set

ROParity:       T ← $RStopBits[RDataMask],                           AT [ROBitLoc,3];
        $RParityDisp[RDataMask];
        DISP [ROParNone], ROData ← T;
ROParNone:      GOTO [ROIdle], RState ← (RState) AND NOT (ROBStateMask),    AT [ROParLoc,0];
ROParOdd:       GOTO [ROParEven], RState ← (RState) XOR (ROParityMask),     AT [ROParLoc,1];
ROParEven:      GOTO [ROParSend], T ← LDF[RState, ROParityBit, 1],          AT [ROParLoc,2];
ROParOne:       GOTO [ROParSend], T ← 1C,                                   AT [ROParLoc,3];
ROParZero:      GOTO [ROParSend], T ← 0C,                                   AT [ROParLoc,4];
ROParSend:      T ← Stack ← (Stack) OR (T);
        RS232 ← T;
        GOTO [ROSetTimer], RState ← (RState) AND NOT (ROBStateMask);

* Idle states, used for idling line and sending break

        Stack ← (Stack) OR (1C),                                   AT [ROBitLoc,4];
        RETURN, STKP ← RStackSave;
        GOTO [ROSetTimer],                                         AT [ROBitLoc,5];
```

```
        SET TASK[RBTask];
        ONPAGE [RBPage];

* Notify frame task that poller has work

RPTurnon:       SKIPON [R<0], RState ← (RState) OR (OR@[RPActive!, RXActive!]C), AT [RFWakeLoc];
        LOADTIMER[RFrameTimer1];
RTimerReturn: NOP;
        RETURN;
```

```
        SET TASK[RFTask];
        ONPAGE [RFPage];

* Frame dispatch code
* The frame code works in the following way:
* Whenever the input bit code, output bit code, or poller code has some work for the
* frame task to perform, it set the appropriate bit in RState indicating work has to
* be done (RIActive, ROActive, or RPActive) and then if the frame code is not currently
* running (RXActive indicates frame code running), it sets RXActive and loads RFrameTimer1
* with a short timer.  When this timer expires, control is passed to the following
* instructions.  They simply do a RETURN with UseCTask set to return to the last place
* the frame code did a TASK.  In addition, they reload the timer, so that after the next
* TASK, the frame code will again get control.  The frame code can then run as though it
* were a normal task doing RETURNS to give control to others.  When it has finished
* processing, it check the RIActive, ROActive, and RPActive bits in RState to see if
* more work needs to be done.  If not, it clears RXActive and idles the frame timer.

        UseCTask, AT [RFWakeLoc];
        RETURN, LOADTIMER[RFrameTimer1];

* The following code is used to initially set registers and the TPC for the frame task.

        SET TASK[0];

        T ← R0, AT[RS232StartLoc];              * Notify frame task at RS232Start1
        R0 ← $RSetDispLo[RFTask,RS232StartLoc1];
        R0 ← (R0) OR ($RSetDispHi[RFTask, RS232StartLoc1]);
        APC&APCTask ← R0;
        RETURN, R0 ← T;         * Restore R0

        SET TASK[RFTask];

        RFrameTimer1 ← AND@[RFShortTimerLo,377]C, AT [RS232StartLoc1];
        RFrameTimer1 ← (RFrameTimer1) OR (AND@[RFShortTimerLo,177400]C);
        RBasePage0 ← 0C;
        RBasePage0Lo ← 0C;
        RState ← 0C;
        RPtrCSB ← AND@[RS232CSBLoc,377]C;
        GOTO [RChkActive], RPtrCSB ← (RPtrCSB) OR (AND@[RS232CSBLoc,177400]C);

* Checks to see if any part of RS232 code desires service.  If so, the appropriate code is
* branched to.  If not, the frame timer is turned off to kill the frame wakeups.

RNoneActive: RTemp2 ← (RTemp2) OR (AND@[RFIdleTimerLo,177400]C);
        LOADTIMER[RTemp2];
        NOP, TASK;
        RState ← (RState) AND NOT (RActiveMask);
RChkActive: LU ← (RState) AND (RIActive);                           * Check input active
        GOTO [RIFrameStart, ALU#0], LU ← (RState) AND (ROActive);  * Check output active
        GOTO [ROFrameStart, ALU#0], LU ← (RState) AND (RPActive);  * Check poller active
        DBLGOTO [RPollerStart, RNoneActive, ALU#0], RTemp2 ← AND@[RFIdleTimerLo,377]C;
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Poller.  Updates ToDCE bits and checks FromDCE bits notifying the user when the bits
* change.  Also, handles commands

RPollerStart: T ← (RPtrCSB) + (RIndexToDCE);
        CALL [RRTask], PFetch2[RBasePage0, RPToDCE];

* Process commands.  If break in progress, shift bit in ROData to the
* right. When it makes ROData ODD, break if finished.  Clear break
* in progress and restart transmitter.

        LU ← (RState) AND (RBreakInProgress);
        GOTO [RPDoCmd, ALU=0], $RPCmdDisp[RPToDCE];
        GOTO [RPCmdE, R>=0], ROData ← (ROData) - 1;
        GOTO [RPStartXmtr], RState ← (RState) AND NOT (RBreakInProgress);
RPDoCmd:          DISP[.+1];

* Commands
* 0 = Command proccessed
* 1 = Nop
* 2 = Reset
* 3 = startReceiver
* 4 = startTransmitter
* 5 = Stop Receiver
* 6 = Stop Transmitter
* 7 = Send Break
* 10 = Set Parameter
* 11 = Clear Ring Indicator
* 12 = Clear break detected
* 13 = Clear Data Lost

* Command = 0 => Command processed
* Don't touch command field in case Mesa is in middle of storing a command.

RPNoCmd:          GOTO [RPCmdE1],                                   AT [RPCmdLoc,0];

* Command = 1 => NOP
* Set command field to zero.  Used to make sure poller is alive.

RPNop:  GOTO [RPCmdE],                                        AT [RPCmdLoc,1];

* Command = 2 => Reset
* NOP for now.   ** DO WE NEED THIS COMMAND? **

RPReset:          GOTO [RPCmdE],                                    AT [RPCmdLoc,2];

* Command = 3 => Start Receiver
* Start receiver timers and set input state to waiting for start bit

RPStartRcvr: LOADPAGE[RBPage],                                   AT [RPCmdLoc,3];
        CALLP[RISetStart];
        GOTO [RPCmdE], RState ← (RState) AND NOT (RIBStateMask);

* Command = 4 => Start transmitter.
* Set transmitter timers and set output state to sending idle.

RPStartXmtr: T ← (RPtrCSB) + (RIndexCharOut),                    AT [RPCmdLoc,4];
        RTemp2 ← 100000C;
        PStore1[RBasePage0, RTemp2];
        LOADPAGE[RBPage];
        CALLP[ROSetTimer1];
        RState ← (RState) AND NOT (ROBStateMask);
        GOTO [RPCmdE], ROData ← 200C;

* Command = 5 => Stop receiver.
* Set idle timer for receiver bit code.

RPStopRcvr: RTemp2 ← AND@[RIIdleTimerLo,177400]C,                AT [RPCmdLoc,5];
        RTemp2 ← (RTemp2) OR (AND@[RIIdleTimerLo,377]C);
        GOTO [RPCmdE], LOADTIMER[RTemp2];

* Command = 6 => Stop transmitter and idle line.
* Set short timer and set line to 1.

RPStopXmtr: RState ← (RState) AND NOT (ROBStateMask),            AT [RPCmdLoc,6];
        RState ← (RState) + (ROBStateIncr4);
        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← Stack ← (Stack) OR (1C);
        STKP ← RStackSave;
        RS232 ← T;
ROShortSet: RTemp2 ← AND@[ROShortTimerLo,177400]C;
        RTemp2 ← (RTemp2) OR (AND@[ROShortTimerLo,377]C);
        GOTO [RPCmdE], LOADTIMER[RTemp2];

* Command = 7 => Send Break
* Set output state to sendbreak, enable break timer, and set short timer

RPBreak:          RState ← (RState) AND NOT (ROBStateMask),          AT [RPCmdLoc,7];
        RState ← (RState) + (ROBStateIncr5);
        ROData ← 62C;
        GOTO [ROShortSet], RState ← (RState) OR (RBreakInProgress);


* Command = 10 => Set Parameters
* Load registers from D0 memory
```

```
RPSetParm:      T ← (RPtrCSB) + (RIndexParm),                              AT [RPCmdLoc,10];
        CALL [RFixBaseReg], PFetch2[RBasePage0, RBufBase];
        PFetch4[RBufBase, RIFullBitLo,0];
        CALL [RRTask], PFetch2[RBufBase, ROFullBitLo,4];
        GOTO [RPCmdE], PFetch1[RBufBase, RDataMask,6];

* Command = 11 => Clear Ring Indicator

RPRing: GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCERing), AT [RPCmdLoc,11];

* Command = 11 => Clear Break Detected

RPNoBreak:      GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCEBreak),AT [RPCmdLoc,12];

* Command = 11 => Clear Data Lost

RPLost: GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCELost), AT [RPCmdLoc,13];

* Clear command field

RPCmdE: RPToDCE ← (RPToDCE) AND (377C);
        T ← (RPtrCSB) + (RIndexToDCE), TASK;
        PStore1[RBasePage0, RPToDCE];

* Set ToDCE bits.

RPCmdE1:        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← (RPToDCE) OR NOT (RToDCEMask);
        Stack ← (Stack) AND NOT (RToDCEMask);
        T ← (Stack) ← (Stack) OR NOT (T);
        RS232 ← T, TASK;
        STKP ← RStackSave;

* Check FromDCE bits and post user if they have changed.  Note that the bits are negative
* logic.  Also, RPNew willl contain any non-hardware bits that have to be set (like
* DataLost, BreakDetected, BreakInProgress).

        T ← (RS232) OR NOT (RFromDCEMask);
        RPNew ← (RPNew) OR NOT (T);
        T ← (RPFromDCE) AND NOT (RNotLatchedMask);
        RPNew ← (RPNew) OR (T);
        T ← (RPtrCSB) + (RIndexFromDCE), TASK;
        PStore1[RBasePage0, RPNew];
        T ← RPFromDCE;
        LU ← (RPNew) XOR (T);
        GOTO [RNoDCEChange, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        NOP;
        CALL [RPost], PFetch1[RBasePage0,RTemp2];
RNoDCEChange: RPNew ← 0C;
        GOTO [RChkActive], RState ← (RState) AND NOT (RPActive);
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Input event, have received a character.  Dispatch on input event.
* Events are:
* 0 => Received a good character.  Store it in IOCB buffer.
* 1 => Received break. Signal Break detected.
* 2 => Received character with parity error.  Set error and store character in IOCB.
* 3 => Received character with frameing error.  Set error and store character in IOCB.

RIFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBIn);
        RFState ← (RFState) OR (RFStateInput);
        CALL [RGetPtrs], UseCTask;
        GOTO [RIDataLost, ALU=0], T ← $RIEventDisp[RData];
        DISP[.+1], RTemp2 ← RStatFrameErr;

* Events are:
* 0 => Character received OK
* 1 => Break received
* 2 => Parity error
* 3 => Framing error

        GOTO [RIStoreChar],                                             AT [RIEventLoc,0];
        GOTO [RIFinish], RPNew ← (RPNew) OR (RFromDCEBreak),            AT [RIEventLoc,1];
        RTemp2 ← RStatParityErr,                                        AT [RIEventLoc,2];
        NOP,                                                            AT [RIEventLoc,3];
        CALL [RSetStatus];

* Character received, store it in buffer.  If not enough room in buffer, advance to next
* IOCB unless RCmdEnd set in which case set DATALOST and wait for frame to end.

RIStoreChar: CALL [RRTask], T ← (RRCount) ← (RRCount) + 1;      * Verify room in buffer
        LU ← (RMaxCount) - T;
        GOTO [RIBufSpace, ALU>=0], LU ← (RCommand) AND (RCmdEnd);
RIBufFull:      GOTO [RIMustEnd, ALU#0], RRCount ← (RRCount) - 1;
        NOP;
        CALL [RDoneIOCB];
        GOTO [RIStoreChar];
RIMustEnd:      CALL [RSetStatus], RTemp2 ← RStatLost;
        GOTO [RIDoBCC];

* Room in buffer.  Store character in buffer.  This is slightly complicated due to the
* fact that we are storing a byte rather than a word.  ROffset+RRCount gives the byte
* offset into the buffer.

RIBufSpace: ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        GOTO [RIStoreEven, R EVEN], T ← RSH[ROffSet,1];
        PFetch1[RBufBase,RTemp2];
        T ← RHMASK[RData], CALL[RRTask];
        RTemp2 ← (LHMASK[RTemp2]) OR T;
        GOTO [RIDoStore];
RIStoreEven: T ← LSH[RData,10];
        RTemp2 ← T;
RIDoStore:      T ←  RSH[ROffSet,1], TASK;
        PStore1[RBufBase,RTemp2];
RIDoBCC:        CALL [RDoBCC];

* Now dispatch on frame state
* States are:
* 0 => just inputted a character, check if special case
* 1 => just inputting first part of BCC, increment state
* 2 => just inputted second half of BCC, check it and end frame
* 3 => counting down to end of frame, if RFState<0 end the frame

        $RFStateDisp[RFState];
        DISP[.+1];

* State = 0.  Normal state, check if special type of character.
* Special types are:
* 0 => not special type.  Do nothing.
* 1 => BCC starts accumulating AFTER this character.  Set RBCC ← 0
* 2 => BCC received AFTER this character.  Increment FState to receiving BCC state.
* 3 => Frame ends in n characters.  Set FState to countdown to end of frame.

        $RICharDisp[RTemp2],                                        AT [RIFrameLoc,0];
        DISP [.+1], RTemp2 ← (LDF[RTemp2,16,2])-1;
        GOTO [RIFinish],                                            AT [RISpecLoc,0];
        GOTO [RIFinish], RBCC ← 0C,                                 AT [RISpecLoc,1];
        GOTO [RIFinish], RFState ← (RFState) + (RFStateIncr),      AT [RISpecLoc,2];
        T ← LSH[RTemp2,14],                                         AT [RISpecLoc,3];
        RFState ← (RFState) + (RFStateIncr3);
        GOTO [RICheckEOF], RFState ← (RFState) OR (T);

* State = 1.  Inputting first half of BCC.  Increment FState

        GOTO [RIFinish], RFState ← (RFState) + (RFStateIncr),      AT [RIFrameLoc,1];

* State = 2.  Got second half of BCC.  Check that accumulated BCC is equal to zero
* and set BCC error if not.  Then end the frame.

        LU ← RBCC,                                                  AT [RIFrameLoc,2];
        GOTO [RIBCCOk, ALU=0];
        NOP;
        CALL [RSetStatus], RTemp2 ← RStatBCCErr;
RIBCCOk:        GOTO [RIEndFrame], RFState ← (RFState) - (RFStateIncr2);

* State=3.  Ending frame after n characters.  Decrement top part of RFState checking it
* first to see if has gone negative.  If it has, end of frame has occurred, so reset
* FState and end the frame.
```

```
RICheckEOF: SKIPON [R<0], RFState ← (RFState) - (RFCountIncr),        AT [RIFrameLoc,3];
        GOTO [RIFinish];
        RFState ← (RFState) - (RFStateIncr3);
RIEndFrame: RFState ← (RFState) AND NOT (RFStateCountBits);
        CALL [RDoneIOCB];
        SKIPON [ALU=0], LU ← (RCommand) AND (RCmdStart);
        DBLGOTO [RIEndFrame, RIFinish, ALU=0];
        GOTO [RINoSave];

* Save registers in IOCB
* Save registers in CSB
* Check for more work

RIFinish:       CALL [RSaveIOCB];
RINoSave:       CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [RChkActive], RState ← (RState) AND NOT (RIActive);

* No input IOCB, mark Data Lost
* Currently DataLost is set when a break comes in without any IOCBS.
* (This is NOT CORRECT!!!!)

RIDataLost: GOTO [RINoSave], RPNew ← (RPNew) OR (RFromDCELost);
```

```
        SET TASK[RFTask];
        ONPAGE [RFPage];

* Output character has been sent.  Load up registers.

ROFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBOut);
        CALL [RGetPtrs], UseCTask;

* Need a character.  Dispatch on state
* States are:
* 0 => Just sent a character.  Get next character from IOCB.
* 1 => Just sent character before BCC. Send first half of BCC.
* 2 => Just sent first half of BCC. Send second half of BCC.
* 3 => Just sent last half of BCC. Idle transmitter.
* 4 => Just idled transmitter.  Signal IOCB complete.

RODispatch: $RFStateDisp[RFState];
        DISP[.+1], T ← RRCount ← (RRCount) - 1;

* A character has been sent, see if any more characters in buffer.
* If nothing in buffer, check EndFrame bit and if set, end the frame.

ROCheckCount: GOTO [ROGotChar, ALU>=0], LU ← (RCommand) AND (RCmdEnd), AT [ROFrameLoc,0];
        GOTO [ROEndFrame,ALU#0], RRCount ← (RRCount) + 1;
RONextIOCB: NOP;
        CALL [RDoneIOCB];
        GOTO [RODispatch];

* Got character, send it.  Again, things are slightly complicated by the fact that we
* are sending a byte rather than a word.

ROGotChar:      T ← (RMaxCount) - T;
        ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        T ← RSH[ROffSet,1], TASK;
        PFetch1[RBufBase, RData];
        GOTO [ROSendOdd, R ODD], LU ← ROffSet;
ROSendEven: GOTO [RODoBCC], RData ← RSH[RData,10];
ROSendOdd:          GOTO [RODoBCC], RData ← RHMASK[RData];

* End of frame, since we got here, we must want to end frame, so set state to idling
* transmitter.

ROEndFrame: RFState ← (RFState) + (RFStateIncr4), CALL[RRTask];
        GOTO [RONoSave], RData ← 100000C;

* Just sent character before BCC, send first BCC character and increment state

ROSendBCC:      T ← RHMASK[RBCC],                                  AT [ROFrameLoc,1];
        RData ← T, CALL [RRTask];
ROIncrFrame: GOTO [RONoSave], RFState ← (RFState) + (RFStateIncr);

* Just sent first half of BCC, send the second half.

        GOTO [ROSendBCC], RBCC ← RSH[RBCC,10],               AT [ROFrameLoc,2];

* Just sent second half of BCC, look for more characters

ROIdleXmtr: RFState ← (RFState) - (RFStateIncr3),             AT [ROFrameLoc,3];
        GOTO [ROCheckCount], LU ← RRCount;

* Just idled transmitter, mark IOCB complete and advance to next IOCB

        GOTO [RONextIOCB],RFState ← (RFState) - (RFStateIncr4),     AT [ROFrameLoc,4];

* Calculate BCC and check for special characters
* Save IOCB registers
* Save CSB registers
* Check for more work

RODoBCC:        CALL [RDoBCC];
        $RICharDisp[RTemp2];
        DISP [.+1];
        GOTO [ROFinish],                                       AT [ROSpecLoc,0];
        GOTO [ROFinish], RBCC ← 0C,                            AT [ROSpecLoc,1];
        GOTO [ROFinish], RFState ← (RFState) + (RFStateIncr),  AT [ROSpecLoc,2];
ROFinish:       CALL [RSaveIOCB],                                      AT [ROSpecLoc,3];
RONoSave:       CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBOut);
        GOTO [RChkActive], RState ← (RState) AND NOT (ROActive);
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Finish off IOCB buffer.  If status is zero, set status to success.
* Dequeue IOCB from chain.  Issue naked notify.
* If new IOCB exists, reload registers from IOCB.

RDoneIOCB:      UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
RDoneIOCB1:LU ← (RComp) AND (RCmpStatus);
        SKIPON [ALU#0];
        RComp ← (RComp) + (RStatOk);
        CALL [RSaveIOCB], RComp ← (RComp) OR (RCmpProc);
        CALL [RNextIOCB], T ← (RPtrIOCB) + (RIndexNext);
        T ← (RCommand) AND (OR@[RCmdWakeAll!,RCmdWakeErr!]C);
        LU ← (RComp) AND T;
        GOTO [RGetIOCB, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        NOP;
        CALL [RPost], PFetch1[RBasePage0, RTemp2];
        GOTO [RGetIOCB];

* Get buffer pointers.

RGetPtrIOCB: RETURN, PFetch4[RBasePage0,RPtrIOCB];
RGetPtrs:       T ← APC&APCTask;
        RSave ← T;
RGetIOCB:       T ← RPtrIOCB;
        GOTO [RNoIOCB, ALU=0];
        CALL [RRTask], PFetch4[RBasePage0,RBufBase];
        T ← (RPtrIOCB) + (RIndexCount);
        CALL [RFixBaseReg], PFetch4[RBasePage0,RRCount];
        GOTO [RSubReturn];

* What to do if we don't have an IOCB!
* On input, return to caller and let him decide if he wants to set DataLost
* On output, idle line

RNoIOCB:        DBLGOTO[RINoIOCB, RONoIOCB, R ODD], LU ← RFState;
RINoIOCB:       GOTO [RSubReturn];
RoNoIOCB:       GOTO [RONoSave], RData ← 100000C;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Accumulate BCC

RDoBCC: UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
        T ← (RPtrCSB) + (RIndexBCCTable);
        CALL [RFixBaseReg], PFetch2[RBasePage0, RBufBase];
        T ← RData;
        T ← (RHMASK[RBCC]) XOR (T), TASK;
        PFetch1[RBufBase,RTemp2];
        T ← (RTemp2) ;
        RBCC ← (RSH[RBCC,10]) XOR T;

* Load character type from endofframe table.
* Types of characters
* 0= normal
* 1=start BCC
* 2=end BCC (ie generate or check BCC).  Implies end of block on input
* 3=end of block

RCheckSpec: T ← (RData) + (400C);
        CALL [RRTask], PFetch1 [RBufBase, RTemp2];
RSubReturn: APC&APCTask ← RSave;
RRTask: RETURN, LU ← RPtrIOCB;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Fix up base register.

RFixBaseReg: T ← (LSH[RBufBaseLo,10]) + 1;
        RETURN, RBufBaseLo ← (RBufBaseLo) OR (T);


* Set completion status.  If no previous error, set status in IOCB and set error.

RSetStatus: LU ← (RComp) AND (RCmpStatus);
        GOTO [RRTask, ALU#0], T ← (RTemp2) OR (RCmpError);
        RETURN, RComp ← (RComp) OR (T);


* Save IOCB ptrs.

RSaveIOCB:      T ← (RPtrIOCB) + (RIndexCount);
        RETURN, PStore4[RBasePage0, RRCount];


* Save CSB ptrs.

RSaveCSB:       PFetch1[RBasePage0, RPtrIOCB];
        LU ← RPtrIOCB;
        RETURN, PStore4[RBasePage0, RPtrIOCB];


* Point to next IOCB.  This has to be done without TASKing.

RNextIOCB:      PFetch1[RBasePage0, RNext];
        DBLGOTO[.+1,.+2, R ODD], LU ← RFState;
        GOTO [.+2], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [.+1], T ← (RPtrCSB) + (RIndexIOCBOut);
        LU ← RNext;
        PStore1[RBasePage0,RNext];
        T ← RNext;
        RETURN, RPtrIOCB ← T;


* Post Complete

RPost:  RStackSave ← pNWW;
        T ← (GetRSpec[103]) XOR (377C); * read stkp
        STKP ← RStackSave, RStackSave ← T, NoRegILockOK;
        T ← RTemp2;
        Stack ← (Stack) OR (T);
        RTemp2 ← pRSImage;
        STKP ← RTemp2;
        T ← Stack ← (Stack) OR (10C);
        RS232 ← T;
        RETURN, STKP ← RStackSave;

END;
```

```
*       Last modified by BRD on June 1, 1979  12:13 PM
*       Version 3.0

insert[RS232Async];
insert[RS232Test];

END;
```

```
insert[d0lang];

NOMIDASINIT;LANGVERSION;

        TITLE [RS232Bit];

* D0 microcode for bit synchronous RS232C
* Last modified by BRD on August 28, 1979  1:12 PM fix GlobalDefs conflicts
* modified by BRD on June 1, 1979  12:13 PM

insert[GlobalDefs];
insert[RS232Defs];

        SET TASK [RBTask];
        ON PAGE [RBPage];

* Timer wakeup occurred on input.   Get input bit and check if zero or one bit

RIWake: T ← (RS232) AND (RI232Data), AT [RIWakeLoc];
        DBLGOTO[RIFoundZero, RIFoundOne, ALU=0], RI1Count ← (RI1Count) - 1;

* Zero bit
*
* RI1Count > 0 ==> normal: shift this bit into character if line in sync
* RI1Count = 0 ==> 5 ones in a row: this is a fill zero so throw it away
* RI1Count < 0 ==> special character, either flag or abort

RIFoundZero: GOTO[RICheckSpec, ALU<0], LU ← (RI1Count);
        DBLGOTO [RIShiftChar, RISetTimer, ALU#0], RI1Count ← 6C;

* One bit
*
* RI1Count > 20 ==> line idle: set count to 21;
* RI1Count > 0 ==> normal: shift this bit into character if line in sync
* RI1Count <= 0 ==> inside special character, ignore bit
* RI1Count = -20 ==> line is idle: signal frame task, and set count to +21

RIFoundOne: GOTO[RICheckIdle, ALU>=0], LU ← (RI1Count) + (20C);
        SKIPON [ALU#0];
        GOTO [RISignalIdle], RI1Count ← 21C;
        GOTO [RTimerReturn], LOADTIMER[RISyncTimer];
RICheckIdle: LU ← (RI1Count) AND (20C);
        GOTO [RIShiftChar1, ALU=0];
        RI1Count ← 21C;
RIShiftChar1: GOTO [RIShiftChar];
RINextChar:RIData ← 200C;
RISetTimer: GOTO [RTimerReturn], LOADTIMER[RISyncTimer];

* Shift in character.

RIShiftChar: SKIPON [R ODD], RIData ← (RSH[RIData,1]) OR (T);
        GOTO [RTimerReturn], LOADTIMER[RISyncTimer];

* Have a character, decide if a valid one
* States are:
* 0 => line idle, ignore character
* 1 => line in sync, notify frame code
* 2 => line in frame, justify character and notify frame code

        $RIBitDisp[RState];
        DISP[.+1];
        GOTO [RINextChar],                                        AT [RIBitLoc,0];
        RState ← (RState) + (RIBStateIncr),                       AT [RIBitLoc,1];
        GOTO [RITurnOn],                                          AT [RIBitLoc,2];

* Special character, determine if idle (RI1Count=-1) or abort (RI1Count<-1)

RICheckSpec: LU ← (RI1Count) + 1;
        DBLGOTO[RISignalFlag, RISignalAbort, ALU=0], RI1Count ← 6C;

* Have a flag.  Dispatch on state
* States are:
* 0 => line idle, increment state
* 1 => line in sync, ignore
* 2 => line in frame, signal endofframe, decrement state

RISignalFlag: $RIBitDisp[RState];
        DISP[.+1];
        GOTO [RINextChar], RState ← (RState) + (RIBStateIncr), AT [RIFlagLoc,0];
        GOTO [RINextChar],                                   AT [RIFlagLoc,1];
        RState ← (RState) - (RIBStateIncr),                  AT [RIFlagLoc,2];
        GOTO [RITurnOn], RIData ← RIDataFlag;

* Have an ABRT.  Signal frame code

RISignalAbort: RState ← (RState) AND NOT (RIBStateMask);
        GOTO [RITurnOn], RIData ← RIDataAbrt;

* Have detected an idle line.  Decide if should notify frame task
* States are:
* 0 => line idle, ignore
* 1 => line in sync, set state to idle
* 2 => line in frame, notify frame task and set state to idle

RISignalIdle: $RIBitDisp[RState];
        DISP [.+1];
        GOTO [RINextChar],                                        AT [RIIdleLoc,0];
        GOTO [RINextChar], RSTate ← (RState) AND NOT (RIBStateMask), AT [RIIdleLoc,1];
        RState ← (RState) AND NOT (RIBStateMask),                 AT [RIIdleLoc,2];
        GOTO[RITurnOn], RIData ← RIDataIdle;
```

```
* Notify frame code that something happened

RITurnOn:       SKIPON [R<0], RState ← (RState) OR (OR@[RIActive!,RXActive!]C);
        LOADTIMER[RFrameTimer1];
        T ← (RPtrCSB) + (RIndexCharIn);
        PStore1[RBasePage0, RIData];
        GOTO [RINextChar];
```

```
        SET TASK [RBTask];
        ON PAGE [RBPage];

* Timer wakeup occurred on output.

ROWake: RStackSave ← pRSImage, AT [ROWakeLoc];
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        $ROBitDisp[RState];
        DISP[.+1],  T ← Stack ← (Stack) AND NOT (1C);

* Send zero and notify task if count < 0.  Otherwise, switch to sending ones

ROSendZero: RS232 ← T,                                          AT [ROBitLoc, 0];
        GOTO [ROTurnon0, R<0], LU ← RO1Count;
        RState ← (RState) + (ROBStateIncr);
ROSetTimer: GOTO [ROStackReset], LOADTIMER[ROSyncTimer];
ROTurnon0:      RState ← (RState) + (ROBStateIncr2);
        GOTO [ROTurnon], RO1Count ← 4C;

* Send one and switch state to SendZero if RO1Count < 0

ROSendOne: T ← Stack ← (Stack) OR (1C),                          AT [ROBitLoc,1];
        GOTO [ROSendIt, R>=0], RO1COunt ← (RO1Count) - 1;
        GOTO [ROSendIt], RState ← (RState) - (ROBStateIncr);


* Send next character bit

        GOTO [ROInsert, R<0], RO1Count ← (RO1Count) - 1,        AT [ROBitLoc, 2];
        T ← LDF[ROData,7,1];
        SKIPON [ALU#0],T ← Stack ← (Stack) OR (T);
        RO1Count ← 4C;
        RS232 ← T;
ROSendBit:      DBLGOTO [ROTurnon, ROSetTimer, R ODD],ROData ← RSH[ROData,1];


* Idle state used to idle line

        GOTO [ROStackReset], Stack ← (Stack) OR (1C),           AT [ROBitLoc, 3];

* Need to insert a zero.  Output zero bit and reset one bit counter

ROInsert:       RO1Count ← 4C;
ROSendIt:       GOTO [ROSetTimer], RS232 ← T;

* Get next character to be sent and notify fame task work needs to be done
* Notify frame task that output has work

ROTurnOn:       LOADTIMER[ROSyncTimer];
        T ← (RPtrCSB) + (RIndexCharOut);
        PFetch1[RBasePage0, ROData];
        T ← 200C;
        GOTO [RONotFlag, R>=0], ROData ← (LSH[ROData,10]) OR (T);
ROSendFlag: RO1Count ← 4C;
        RState ← (RState) - (ROBStateIncr2);
RONotFlag: SKIPON [R<0], RState ← (RState) OR (OR@[ROActive!, RXActive!]C);
        LOADTIMER[RFrameTimer1];
ROStackReset: GOTO [RTimerReturn], STKP ← RStackSave;
```

```
        SET TASK[RBTask];
        ONPAGE [RFPage];

* Notify frame task that poller has work

RPTurnon:       SKIPON [R<0], RState ← (RState) OR (OR@[RPActive!, RXActive!]C), AT [RPWakeLoc];
        LOADTIMER[RFrameTimer1];
RTimerReturn: NOP;
        RETURN;
```

```
        SET TASK[RFTask];
        ONPAGE [RFPage];

* Frame dispatch code
* The frame code works in the following way:
* Whenever the input bit code, output bit code, or poller code has some work for the
* frame task to perform, it set the appropriate bit in RState indicating work has to
* be done (RIActive, ROActive, or RPActive) and then if the frame code is not currently
* running (RXActive indicates frame code running), it sets RXActive and loads RFrameTimer1
* with a short timer. When this timer expires, control is passed to the following
* instructions. They simply do a RETURN with UseCTask set to return to the last place
* the frame code did a TASK. In addition, they reload the timer, so that after the next
* TASK, the frame code will again get control. The frame code can then run as though it
* were a normal task doing RETURNS to give control to others. When it has finished
* processing, it check the RIActive, ROActive, and RPActive bits in RState to see if
* more work needs to be done. If not, it clears RXActive and idles the frame timer.

        UseCTask, AT [RFWakeLoc];
        RETURN, LOADTIMER[RFrameTimer1];

* The following code is used to initially set registers and the TPC for the frame task.

        SET TASK[0];

        T ← R0, AT[RS232StartLoc];              * Notify task 0 at RS232Start1
        R0 ← $RSetDispLo[RFTask,RS232StartLoc1];
        R0 ← (R0) OR ($RSetDispHi[RFTask, RS232StartLoc1]);
        APC&APCTask ← R0;
        RETURN, R0 ← T;         * Restore R0

        SET TASK[RFTask];

        RFrameTimer1 ← AND@[RFShortTimerLo,377]C, AT [RS232StartLoc1];
        RFrameTimer1 ← (RFrameTimer1) OR (AND@[RFShortTimerLo,177400]C);
        RBasePage0 ← 0C;
        RBasePage0Lo ← 0C;
        RState ← 0C;
        RPtrCSB ← AND@[RS232CSBLoc,377]C;
        GOTO [RChkActive], RPtrCSB ← (RPtrCSB) OR (AND@[RS232CSBLoc,177400]C);

* Checks to see if any part of RS232 code desires service. If so, it is branched to.
* If not, the frame timer is turned off to kill the frame wakeups.

RNoneActive: RTemp2 ← (RTemp2) OR (AND@[RFIdleTimerLo,177400]C);
        LOADTIMER[RTemp2];
        NOP, TASK;
        RState ← (RState) AND NOT (RActiveMask);
RChkActive: LU ← (RState) AND (RIActive);                         * Check input active
        GOTO [RIFrameStart, ALU#0], LU ← (RState) AND (ROActive);  * Check output active
        GOTO [ROFrameStart, ALU#0], LU ← (RState) AND (RPActive);  * Check poller active
        DBLGOTO [RPollerStart, RNoneActive, ALU#0], RTemp2 ← AND@[RFIdleTimerLo,377]C;
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Poller.  Updates ToDCE bits and checks FromDCE bits notifying the user when the bits
* change.  Also, handles commands

RPollerStart: T ← (RPtrCSB) + (RIndexToDCE);
        CALL [RRTask], PFetch2[RBasePage0, RPToDCE];

* Process commands

        $RPCmdDisp[RPToDCE];
        DISP[RPNop];

* Commands
* 0 = Command proccessed
* 1 = Nop
* 2 = Reset
* 3 = startReceiver
* 4 = startTransmitter
* 5 = Stop Receiver
* 6 = Stop Transmitter
* 7 = Send Break
* 10 = Set Parameter
* 11 = Clear Ring Indicator
* 12 = Clear break detected
* 13 = Clear Data Lost

* Command = 0 => Command processed
* Don't even clear out command field

RPNop:  GOTO [RPCmdE1],                                             AT [RPCmdLoc,0];

* Command = 1 => NOP
* Clear out command field

        GOTO [RPCmdE],                                              AT [RPCmdLoc,1];

* Command = 2 => Reset
* NOP for now

        GOTO [RPCmdE],                                              AT [RPCmdLoc,2];

* Command = 3 => Start Receiver
* Start receiver timers and set input state to waiting for synchronization

        RISyncTimer ← AND@[RISyncTimerLo,177400]C,                  AT [RPCmdLoc,3];
        RISyncTimer ← (RISyncTimer) OR (AND@[RISyncTimerLo,377]C);
        LOADTIMER[RISyncTimer];
        RI1Count ← 21C;
        T ← (RPtrCSB) + (RIndexBCCIn);
        RTemp2 ← (ZERO) - 1;
        PStore1[RBasePage0, RTemp2];
        GOTO [RPCmdE], RState ← (RState) AND NOT (RIBStateMask);

* Command = 4 => Start transmitter
* Set transmitter timers and set output state to sending idle

        T ← (RPtrCSB) + (RIndexCharOut),                            AT [RPCmdLoc,4];
        RTemp2 ← 100000C;
        PStore1[RBasePage0, RTemp2];
        ROSyncTimer ← AND@[ROSyncTimerLo,177400]C;
        ROSyncTimer ← (ROSyncTimer) OR (AND@[ROSyncTimerLo,377]C);
        LOADTIMER [ROSyncTimer];
        RState ← (RState) AND NOT (ROBStateMask);
        RO1Count ← 4C;
        T ← (RPtrCSB) + (RIndexBCCOut);
        RTemp2 ← (ZERO) - 1;
        PStore1[RBasePage0, RTemp2];
        GOTO [RPCmdE], ROData ← 200C;

* Command = 5 => Stop receiver
* Set idle timer

        RTemp2 ← AND@[RIIdleTimerLo,177400]C,                       AT [RPCmdLoc,5];
        RTemp2 ← (RTemp2) OR (AND@[RIIdleTimerLo,377]C);
        GOTO [RPCmdE], LOADTIMER[RTemp2];

* Command = 6 => Stop transmitter and idle line
* Set short timer and set line to 1

        RTemp2 ← AND@[ROShortTimerLo,177400]C,                      AT [RPCmdLoc,6];
        RTemp2 ← (RTemp2) OR (AND@[ROShortTimerLo,377]C);
        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← Stack ← (Stack) OR (1C);
        RS232 ← T;
        LOADTIMER[RTemp2];
        STKP ← RStackSave;
        RState ← (RState) AND NOT (ROBStateMask);
        GOTO [RPCmdE], RState ← (RState) + (ROBStateIncr3);

* Command = 7 => Send Break
* Unimplemented

        GOTO [RPCmdE],                                              AT [RPCmdLoc,7];

* Command = 10 => Set Parameters
* NO PARAMETERS YET!
```

```
        GOTO [RPCmdE],                                              AT [RPCmdLoc,10];

* Command = 11 => Clear Ring Indicator

        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCERing), AT [RPCmdLoc,11];

* Command = 11 => Clear Ring Indicator

        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCEBreak),AT [RPCmdLoc,12];

* Command = 11 => Clear Ring Indicator

        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCELost), AT [RPCmdLoc,13];

* Clear command field if not already cleared

RPCmdE: RPToDCE ← (RPToDCE) AND (377C);
        T ← (RPtrCSB) + (RIndexToDCE), TASK;
        PStore1[RBasePage0, RPToDCE];

* Set ToDCE bits

RPCmdE1:        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← (RPToDCE) OR NOT (RToDCEMask);
        Stack ← (Stack) AND NOT (RToDCEMask);
        T ← (Stack) ← (Stack) OR NOT (T);
        RS232 ← T, TASK;
        STKP ← RStackSave;

* Check FromDCE bits and post user if they have changed.  Note that the bits are negative
* logic.  Also, RPNew willl contain any non-hardware bits that have to be set (like
* DataLost, AbortDetected).

        T ← (RS232) OR NOT (RFromDCEMask);        * Get FromDCE bits
        RPNew ← (RPNew) OR NOT (T);
        T ← (RPFromDCE) AND NOT (RNotLatchedMask); * Save ON lathced bits
        RPNew ← (RPNew) OR (T);
        T ← (RPtrCSB) + (RIndexFromDCE), TASK;    * Store new FromDCE bits;
        PStore1[RBasePage0, RPNew];
        T ← RPFromDCE;                      * Get old FromDCE bits
        LU ← (RPNew) XOR (T);               * Compare to new FromDCE bits
        GOTO [RNoDCEChange, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        NOP;
        CALL [RPost], PFetch1[RBasePage0,RTemp2];
RNoDCEChange: RPNew ← 0C;
        GOTO [RChkActive], RState ← (RState) AND NOT (RPActive);
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Input character received.  Load registers.

RIFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBIn);
        RFState ← (RFState) OR (RFStateInput);
        CALL [RGetPtrs], UseCTask;

* Dispatch on input event.  Test state of input line
* Events are:
* 0 = character received
* 1 = ABRT received
* 2 = FLAG received (end of frame)
* 3 = line returned to idle state (more than 17 ones in a row)

        DBLGOTO [RIDataLost, RIStoreChar, ALU=0], $RIEventDisp[RData];
RIStoreChar: DISP[.+1];

* Character received, store it in IOCB

        T ← (RRCount) ← (RRCount) + 1,                              AT [RIFrameLoc,0];
        LU ← (RMaxCount) - T;
        GOTO [RIBufSpace, ALU>=0], LU ← (RCommand) AND (RCmdEnd);
RIBufFull:       GOTO [RIMustEnd, ALU#0], RRCount ← (RRCount) - 1;
        NOP;
        CALL [RDoneIOCB];
        DBLGOTO [RIDataLost, RIStoreChar, ALU=0], $RIEventDisp[RData];
RIMustEnd:       CALL [RSetStatus], RTemp2 ← RStatLost;
        GOTO [RIDoBCC];

* Room in buffer.  Store character in buffer.  This is slightly complicated due to the
* fact that we are storing a byte rather than a word.  ROffset+RRCount gives the byte
* offset into the buffer.  Update BCC.

RIBufSpace: ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        GOTO [RIStoreEven, R EVEN], T ← RSH[ROffSet,1];
        PFetch1[RBufBase,RTemp2];
        T ← RHMASK[RData], TASK;
        RTemp2 ← (LHMASK[RTemp2]) OR T;
        GOTO [RIDoStore];
RIStoreEven: T ← LSH[RData,10];
        RTemp2 ← T;
RIDoStore:       T ← RSH[ROffSet,1], TASK;
        PStore1[RBufBase,RTemp2];
        GOTO [RIDoBCC];

* ABRT received, end frame and signal error

        GOTO [RIAborted], RTemp2 ← RStatBadFrame,                   AT [RIFrameLoc,1];
RIAborted:       CALL [RSetStatus];
RIEndFrame: NOP;
        CALL[RDoneIOCB];
        SKIPON [ALU=0], LU ← (RCommand) AND (RCmdStart);
        DBLGOTO [RIEndFrame, RINoSave1, ALU=0];
        GOTO [RINoSave];
RINoSave1: GOTO [RINoSave];

* FLAG received, Check BCC and end of frame.

        RBCC ← (RBCC) - (AND@[RBCCRemainder,177400]C),              AT [RIFrameLoc,2];
        RBCC ← (RBCC) - (AND@[RBCCRemainder,377]C);
        SKIPON [ALU=0], RBCC ← (ZERO) - 1;
        GOTO [RIAborted], RTemp2 ← RStatBCCErr;
        GOTO [RIEndFrame];

* Idle line, Tell the user

        GOTO [RIAborted], RTemp2 ← RStatBadFrame,                   AT [RIFrameLoc,3];

* End of frame work

RIDoBCC:         CALL [RDoBCC];
RIFinish:        CALL [RSaveIOCB];
RINoSave:        CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [RChkActive], RState ← (RState) AND NOT (RIActive);

* Data arrived and no IOCB on chain.  Signal Data Lost.

RIDataLost: GOTO [RINoSave], RPNew ← (RPNew) OR (RFromDCELost);
```

```
        SET TASK [RFTask];
        ON PAGE [RFPage];

* Software USRT needs another character. Load registers

ROFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBOut);
        CALL [RGetPtrs], UseCTask;

* Dispatch on output state.
* States are
* 0 = > Sent a character/opening flag
* 1 = > Sent BCC1
* 2 = > Sent BCC2
* 3 = > Sent closing/opening flag

RODispatch: $RFStateDisp[RFState];
        DISP[.+1], T ← RRCount ← (RRCount) - 1;

* A character has been sent, see if any more characters in buffer.
* If nothing in buffer, check EndFrame bit and if set, end the frame.

ROCheckCount: GOTO [ROGotChar, ALU>=0], LU ← (RCommand) AND (RCmdEnd),  AT [ROFrameLoc,0];
        GOTO [ROEndFrame, ALU#0], RRCount ← (RRCount) + 1;
RONextIOCB: NOP;
        CALL [RDoneIOCB];
        GOTO [RODispatch];

* Got character, send it.  Again, things are slightly complicated by the fact that we
* are sending a byte rather than a word.

ROGotChar:      T ← (RMaxCount) - T;
        ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        T ← RSH[ROffSet,1], TASK;
        PFetch1[RBufBase, RData];
        GOTO [ROSendOdd, R ODD], LU ← ROffSet;
ROSendEven: GOTO [RODoBCC], RData ← RSH[RData,10];
ROSendOdd:          GOTO [RODoBCC], RData ← RHMASK[RData];

* End of frame, send first BCC character

ROEndFrame: T ← RHMASK[RBCC];
ROSendBCC:      RData ← (ZERO) OR NOT (T);
        RData ← RHMASK[RData];
ROIncrFrame: GOTO [ROFinish], RFState ← (RFState) + (RFStateIncr);

* Send second BCC character

        T ← RSH[RBCC,10],                                       AT [ROFrameLoc,1];
        GOTO [ROSendBCC], RBCC ← (ZERO) - 1;

* Send a closing flag

        GOTO [ROIncrFrame], RData ← 100000C,                    AT [ROFrameLoc,2];

* Just sent closing flag, mark IOCB complete and advance to next IOCB

        GOTO [RONextIOCB], RFState ← (RFState) - (RFStateIncr3),     AT [ROFrameLoc,3];

* Finish up with character by doing BCC calculation.
* Save registers in IOCB
* Save registers in CSB
* Check for more work

RODoBCC:        CALL [RDoBCC];
ROFinish:       CALL [RSaveIOCB];
RONoSave:       CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBOut);
        GOTO [RChkActive], RState ← (RState) AND NOT (ROActive);
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Finish off IOCB buffer

RDoneIOCB:      UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
RDoneIOCB1:LU ← (RComp) AND (RCmpStatus);
        SKIPON [ALU#0];
        RComp ← (RComp) + (RStatOk);
        CALL [RSaveIOCB], RComp ← (RComp) OR (RCmpProc);
        CALL [RNextIOCB], T ← (RPtrIOCB) + (RIndexNext);
        T ← (RCommand) AND (OR@[RCmdWakeAll!,RCmdWakeErr!]C);
        LU ← (RComp) AND T;
        GOTO [RGetIOCB, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        NOP;
        CALL [RPost], PFetch1[RBasePage0, RTemp2];
        GOTO [RGetIOCB];

* Get buffer pointers.

RGetPtrIOCB: RETURN, PFetch4[RBasePage0,RPtrIOCB];
RGetPtrs:       T ← APC&APCTask;
        RSave ← T;
RGetIOCB:       T ← RPtrIOCB;
        GOTO [RNoIOCB, ALU=0];
        CALL [RRTask], PFetch4[RBasePage0,RBufBase];
        T ← (RPtrIOCB) + (RIndexCount);
        CALL [RFixBaseReg], PFetch4[RBasePage0,RRCount];
        GOTO [RSubReturn];

* What to do if we don't have an IOCB!
* On input, signal data lost (NOT IMPLEMENTED YET)
* On output, send a flag

RNoIOCB:        DBLGOTO[RINoIOCB, RONoIOCB, R ODD], LU ← RFState;
RINoIOCB:       GOTO [RSubReturn];
RoNoIOCB:       GOTO [RONoSave], RData ← 100000C;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Accumulate BCC

RDoBCC: UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
        T ← (RPtrCSB) + (RIndexBCCTable);
        CALL [RFixBaseReg], PFetch2[RBasePage0, RBufBase];
        T ← RData;
        T ← (RHMASK[RBCC]) XOR (T), TASK;
        PFetch1[RBufBase,RTemp2];
        T ← (RTemp2) ;
        RBCC ← (RSH[RBCC,10]) XOR T;
RSubReturn: APC&APCTask ← RSave;
RRTask: RETURN, LU ← RPtrIOCB;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Fix up base register

RFixBaseReg: T ← (LSH[RBufBaseLo,10]) + 1;
        RETURN, RBufBaseLo ← (RBufBaseLo) OR (T);


* Set completion status

RSetStatus: LU ← (RComp) AND (RCmpStatus);
        GOTO [RRTask, ALU#0], T ← (RTemp2) OR (RCmpError);
        RETURN, RComp ← (RComp) OR (T);


* Save IOCB ptrs

RSaveIOCB:      T ← (RPtrIOCB) + (RIndexCount);
        RETURN, PStore4[RBasePage0, RRCount];


* Save CSB ptrs

RSaveCSB:       PFetch1[RBasePage0, RPtrIOCB];
        LU ← RPtrIOCB;
        RETURN, PStore4[RBasePage0, RPtrIOCB];


* Point to next IOCB

RNextIOCB:      PFetch1[RBasePage0, RNext];
        DBLGOTO[.+1,.+2, R ODD], LU ← RFState;
        GOTO [.+2], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [.+1], T ← (RPtrCSB) + (RIndexIOCBOut);
        LU ← RNext;
        PStore1[RBasePage0,RNext];
        T ← RNext;
        RETURN, RPtrIOCB ← T;

RPost:  RStackSave ← pNWW;
        T ← (GetRSpec[103]) XOR (377C); * read stkp
        STKP ← RStackSave, RStackSave ← T, NoRegILockOK;
        T ← RTemp2;
        Stack ← (Stack) OR (T);
        RTemp2 ← pRSImage;
        STKP ← RTemp2;
        T ← Stack ← (Stack) OR (10C);
        RS232 ← T;
        RETURN, STKP ← RStackSave;

        END;
```

```
*       Last modified by BRD on June 1, 1979  12:13 PM
*       Version 3.0

insert[RS232Bit];
insert[RS232Test];

END;
```

```
insert[d0lang];

        NOMIDASINIT;LANGVERSION;

        TITLE [RS232Byte];

* D0 microcode for byte synchronous
* Last modified by BRD on August 28, 1979  1:09 PM fix GlobalDefs confilicts
* modified by BRD on June 1, 1979  12:11 PM

insert[GlobalDefs];
insert[RS232Defs];

* This microcode uses a timer to simulate a hardware wakeup for the frame task.  It uses
* Task 4 for the frame task, and task 5 for the bit task

        SET TASK [RBTask];
        ONPAGE [RBPage];

* Wakeup occurred on input timer

RIWake: $RIBitDisp[RState], AT [RIWakeLoc];
        DISP[RIWaitSyn], T ← (RS232) AND (RI232Data);

* Waiting for first sync character.  Shift in a bit and compare last 8 bits to
* SYN character.  If a match, we are in sync.

RIWaitSyn:      RIData ← (RSH[RIData,1]) OR (T),                        AT [RIBitLoc,0];
        T ← RHMASK[RIData];
        LU ← (RSyncChar) - (T);
        GOTO [RISetTimer, ALU#0];
        GOTO [RINextChar], RState ← (RState) + (RIBStateIncr);

* Data bit in.  Shift into partially assembled character.  If end of character, justify
* character and check if parity bit coming

RIShiftBit: RState ← (RState) XOR (T),                              AT [RIBitLoc,1];
        SKIPON[R ODD], RIData ← (RSH[RIData,1]) OR (T);
        GOTO [RISetTimer];
        $RIJustDisp[RDataMask];
        DISP [RIJustify];
RIJustify:      GOTO [RIChkPar], RIData ← RSH[RIData,3],                AT [RIJustLoc,0];
        GOTO [RIChkPar], RIData ← RSH[RIData,2],                  AT [RIJustLoc,1];
        GOTO [RIChkPar], RIData ← RSH[RIData,1],                  AT [RIJustLoc,2];
RIChkPar:       LU ← (RDataMask) AND (RMaskParity),                     AT [RIJustLoc,3];
        GOTO [RITurnOn, ALU=0], RState ← (RState) + (RIBStateIncr);
        GOTO [RISetTimer];

* Chock parity is enabled

RIParity:       RState ← (RState) XOR (T),                              AT [RIBitLoc,2];
        $RParityDisp[RDataMask];
        DISP [RIParNone], RBTemp1 ← T;
RIParNone:      GOTO [RITurnOn], RState ← (RState) + (RIBStateIncr),    AT [RIParLoc,0];
RIParOdd:       GOTO [RIParChk], LU ← LDF[RState, RIParityBit, 1],      AT [RIParLoc,1];
RIParEven:      GOTO [RIParChk], LU ← (LDF[RState, RIParityBit, 1]) - 1, AT [RIParLoc,2];
RIParOne:       GOTO [RIParChk], LU ← LDF[RBTemp1, RIParityBit, 1],     AT [RIParLoc,3];
RIParZero:      GOTO [RIParChk], LU ← (LDF[RBTemp1, RIparityBit, 1]) - 1, AT [RIParLoc,4];
RIparChk:       GOTO [RIParDone, ALU#0];
        RIData ← (RIData) OR (RIDataParityErr);
RIParDone:      GOTO [RITurnOn];

* Notify frame task that input has work

RITurnOn:       SKIPON [R<0], RState ← (RState) OR (OR@[RIActive!, RXActive!]C);
        LOADTIMER[RFrameTimer1];
        RState ← (RState) - (RIBStateIncr);
        T ← (RPtrCSB) + (RIndexCharIn);
        PStore1[RBasePage0, RIData];
RINextChar: T ← (RDataMask) AND (RDataSizeMask);
        RIData ← T;
RISetTimer: GOTO [RTimerReturn], LOADTIMER[RISyncTimer];
```

```
        SET TASK[RBTask];
        ONPAGE [RBPage];

* Timer wakeup occurred on output.

ROWake: RStackSave ← pRSImage, AT [ROWakeLoc];
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        $ROBitDisp[RState];
        DISP[ROIdle],  T ← Stack ← (Stack) AND (376C);


* Idle output state.

ROIdle: GOTO [ROSendBit], T ← Stack ← (Stack) OR (1C),              AT [ROBitLoc,0];

* Get next input character.  Notify frame code.
* If we have a character, send it.  If no character to be sent, idle line

ROStart:        SKIPON [R<0], RState ← (RState) OR (OR@[ROActive!,RXActive!]C),AT [ROBitLoc,1];
        LOADTIMER[RFrameTimer1];
        T ← (RPtrCSB) + (RIndexCharOut);
        PFetch1[RBasePage0, ROData];
        T ← (RDataMask) AND (RDataSizeMask);
        SKIPON [R>=0], ROData ← (LSH[ROData,10]) OR (T);
        GOTO [ROIdle], RState ← (RState) - (ROBStateIncr);
        GOTO [ROSendChar], RState ← (RState) + (ROBStateIncr);

* Send data bit

ROSendChar: T ← LDF[ROData,7,1],                                         AT [ROBitLoc,2];
        RState ← (RState) XOR (T);
        T ← Stack ← (Stack) OR (T);
ROSendBit:      RS232 ← T;
        DBLGOTO [RONextState, ROSetTimer, R ODD], ROData ← RSH[ROData,1];
RONextState: RState ← (RState) + (ROBStateIncr);
ROSetTimer: LOADTIMER[ROSyncTimer];
ROStackReset: GOTO [RTimerReturn], STKP ← RStackSave;

* Send parity bit if parity set

ROParity:       $RParityDisp[RDataMask],                                 AT [ROBitLoc,3];
        DISP [ROParNone], RState ← (RState) - (ROBStateIncr2);
ROParNone:      GOTO [ROStart],                                          AT [ROParLoc,0];
ROParOdd:       GOTO [ROParEven], RState ← (RState) XOR (ROParityMask),  AT [ROParLoc,1];
ROParEven:      GOTO [ROParSend], T ← LDF[RState, ROParityBit, 1],       AT [ROParLoc,2];
ROParOne:       GOTO [ROParSend], T ← 1C,                                AT [ROParLoc,3];
ROParZero:      GOTO [ROParSend], T ← 0C,                                AT [ROParLoc,4];
ROParSend:      T ← Stack ← (Stack) OR (T);
        GOTO [ROSetTimer], RS232 ← T;

* State used by poller to stop transmitter.

        GOTO [ROStackReset], Stack ← (Stack) OR (1C),               AT [ROBitLoc,4];
```

```
        SET TASK[RBTask];
        ONPAGE [RFPage];

* Notify frame task that poller has work

RPTurnon:       SKIPON [R<0], RState ← (RState) OR (OR@[RPActive!, RXActive!]C), AT [RPWakeLoc];
        LOADTIMER[RFrameTimer1];
RTimerReturn: NOP;
        RETURN;
```

```
        SET TASK[RFTask];
        ONPAGE [RFPage];

* Frame dispatch code
* The frame code works in the following way:
* Whenever the input bit code, output bit code, or poller code has some work for the
* frame task to perform, it set the appropriate bit in RState indicating work has to
* be done (RIActive, ROActive, or RPActive) and then if the frame code is not currently
* running (RXActive indicates frame code running), it sets RXActive and loads RFrameTimer1
* with a short timer.  When this timer expires, control is passed to the following
* instructions.  They simply do a RETURN with UseCTask set to return to the last place
* the frame code did a TASK.  In addition, they reload the timer, so that after the next
* TASK, the frame code will again get control.  The frame code can then run as though it
* were a normal task doing RETURNS to give control to others.  When it has finished
* processing, it check the RIActive, ROActive, and RPActive bits in RState to see if
* more work needs to be done.  If not, it clears RXActive and idles the frame timer.

        UseCTask, AT [RFWakeLoc];
        RETURN, LOADTIMER[RFrameTimer1];

* The following code is used to initially set registers and the TPC for the frame task.

        SET TASK[0];

        T ← R0, AT[RS232StartLoc];              * Notify task 0 at RS232Start1
        R0 ← $RSetDispLo[RFTask,RS232StartLoc1];
        R0 ← (R0) OR ($RSetDispHi[RFTask, RS232StartLoc1]);
        APC&APCTask ← R0;
        RETURN, R0 ← T;          * Restore R0

        SET TASK[RFTask];

        RFrameTimer1 ← AND@[RFShortTimerLo,377]C, AT [RS232StartLoc1];
        RFrameTimer1 ← (RFrameTimer1) OR (AND@[RFShortTimerLo,177400]C);
        RBasePage0 ← 0C;
        RBasePage0Lo ← 0C;
        RState ← 0C;
        RPtrCSB ← AND@[RS232CSBLoc,377]C;
        GOTO [RChkActive], RPtrCSB ← (RPtrCSB) OR (AND@[RS232CSBLoc,177400]C);

* Checks to see if any part of RS232 code desires service.  If so, it is branched to.
* If not, the frame timer is turned off to kill the frame wakeups.

RNoneActive: RTemp2 ← (RTemp2) OR (AND@[RFIdleTimerLo,177400]C);
        LOADTIMER[RTemp2];
        NOP, TASK;
        RState ← (RState) AND NOT (RActiveMask);
RChkActive: LU ← (RState) AND (RIActive);                           * Check input active
        GOTO [RIFrameStart, ALU#0], LU ← (RState) AND (ROActive);  * Check output active
        GOTO [ROFrameStart, ALU#0], LU ← (RState) AND (RPActive);  * Check poller active
        DBLGOTO [RPollerStart, RNoneActive, ALU#0], RTemp2 ← AND@[RFIdleTimerLo,377]C;
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Poller.  Updates ToDCE bits and checks FromDCE bits notifying the user when the bits
* change.  Also, handles commands

RPollerStart: T ← (RPtrCSB) + (RIndexToDCE);
        CALL [RRTask], PFetch2[RBasePage0, RPToDCE];

* Process commands

        $RPCmdDisp[RPToDCE];
        DISP[RPNop];

* Commands
* 0 = Command proccessed
* 1 = Nop
* 2 = Reset
* 3 = startReceiver
* 4 = startTransmitter
* 5 = Stop Receiver
* 6 = Stop Transmitter
* 7 = Send Break
* 10 = Set Parameter
* 11 = Clear Ring Indicator
* 12 = Clear break detected
* 13 = Clear Data Lost

* Command = 0 => Command processed
* Don't even clear out command field

RPNop:  GOTO [RPCmdE1],                                         AT [RPCmdLoc,0];

* Command = 1 => NOP
* Clear out command field

        GOTO [RPCmdE],                                          AT [RPCmdLoc,1];

* Command = 2 => Reset
* NOP for now

        GOTO [RPCmdE],                                          AT [RPCmdLoc,2];

* Command = 3 => Start Receiver
* Start receiver timers and set input state to waiting for synchronization

        RISyncTimer ← AND@[RISyncTimerLo,177400]C,              AT [RPCmdLoc,3];
        RISyncTimer ← (RISyncTimer) OR (AND@[RISyncTimerLo,377]C);
        LOADTIMER[RISyncTimer];
        GOTO [RPCmdE], RState ← (RState) AND NOT (RIBStateMask);

* Command = 4 => Start transmitter
* Set transmitter timers and set output state to sending idle

        T ← (RPtrCSB) + (RIndexCharOut),                        AT [RPCmdLoc,4];
        RTemp2 ← 100000C;
        PStore1[RBasePage0, RTemp2];
        ROSyncTimer ← AND@[ROSyncTimerLo,177400]C;
        ROSyncTimer ← (ROSyncTimer) OR (AND@[ROSyncTimerLo,377]C);
        LOADTIMER [ROSyncTimer];
        RState ← (RState) AND NOT (ROBStateMask);
        GOTO [RPCmdE], ROData ← 200C;

* Command = 5 => Stop receiver
* Set idle timer

        RTemp2 ← AND@[RIIdleTimerLo,177400]C,                    AT [RPCmdLoc,5];
        RTemp2 ← (RTemp2) OR (AND@[RIIdleTimerLo,377]C);
        GOTO [RPCmdE], LOADTIMER[RTemp2];

* Command = 6 => Stop transmitter and idle line
* Set short timer and set line to 1

        RTemp2 ← AND@[ROShortTimerLo,177400]C,                    AT [RPCmdLoc,6];
        RTemp2 ← (RTemp2) OR (AND@[ROShortTimerLo,377]C);
        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← Stack ← (Stack) OR (1C);
        RS232 ← T;
        STKP ← RStackSave;
        LOADTIMER[RTemp2];
        RState ← (RState) AND NOT (ROBStateMask);
        GOTO [RPCmdE], RState ← (RState) + (ROBStateIncr4);

* Command = 7 => Send Break
* Unimplemented

        GOTO [RPCmdE],                                          AT [RPCmdLoc,7];

* Command = 10 => Set Parameters
* Load registers from D0 memory

        T ← (RPtrCSB) + (RIndexParm),                           AT [RPCmdLoc,10];
        CALL [RFixBaseReg], PFetch2[RBasePage0, RBufBase];
        PFetch1[RBufBase, RDataMask,6];
        GOTO [RPCmdE], PFetch1[RBufBase, RSyncChar,7];

* Command = 11 => Clear Ring Indicator
```

```
        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCERing), AT [RPCmdLoc,11];

* Command = 11 => Clear Ring Indicator

        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCEBreak),AT [RPCmdLoc,12];

* Command = 11 => Clear Ring Indicator

        GOTO [RPCmdE], RPFromDCE ← (RPFromDCE) AND NOT (RFromDCELost), AT [RPCmdLoc,13];

* Clear command field if not already cleared

RPCmdE: RPToDCE ← (RPToDCE) AND (377C);
        T ← (RPtrCSB) + (RIndexToDCE), TASK;
        PStore1[RBasePage0, RPToDCE];

* Set ToDCE bits

RPCmdE1:        RStackSave ← pRSImage;
        T ← (GetRSpec[103]) XOR (377C);
        RStackSave ← T, STKP ← RStackSave, NoRegILockOK;
        T ← (RPToDCE) OR NOT (RToDCEMask);
        Stack ← (Stack) AND NOT (RToDCEMask);
        T ← (Stack) ← (Stack) OR NOT (T);
        RS232 ← T, TASK;
        STKP ← RStackSave;

* Check FromDCE bits and post user if they have changed.  Note that the bits are negative
* logic.  Also, RPNew will contain any non-hardware bits that have to be set (like
* DataLost, BreakDetected).

        T ← (RS232) OR NOT (RFromDCEMask);      * Get FromDCE bits
        RPNew ← (RPNew) OR NOT (T);
        T ← (RPFromDCE) AND NOT (RNotLatchedMask); * Save ON lathced bits
        RPNew ← (RPNew) OR (T);
        T ← (RPtrCSB) + (RIndexFromDCE), TASK;  * Store new FromDCE bits;
        PStore1[RBasePage0, RPNew];
        T ← RPFromDCE;                          * Get old FromDCE bits
        LU ← (RPNew) XOR (T);                   * Compare to new FromDCE bits
        GOTO [RNoDCEChange, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        LoadPage[RBPage];
        CALL [RPost], PFetch1[RBasePage0,RTemp2];
RNoDCEChange: RPNew ← 0C;
        GOTO [RChkActive], RState ← (RState) AND NOT (RPActive);
```

```
        SET TASK[RFTask];
        ON PAGE[RFPage];

* Input event, have received a character.  Dispatch on input event.
* Events are:
* 0 => Received a good character.  Store it in IOCB buffer.
* 1 => Received break. NOT IMPLEMENTED YET.
* 2 => Received character with parity error.  Set error and store character in IOCB.
* 3 => Received character with frameing error.  Set error and store character in IOCB.

RIFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBIn);
        RFState ← (RFState) OR (RFStateInput);
        CALL [RGetPtrs], UseCTask;
        GOTO [RIDataLost, ALU=0],$RIEventDisp[RData];
        DISP[.+1], RTemp2 ← RStatFrameErr;

        GOTO [RIStoreChar],       AT [RIEventLoc,0];
        GOTO [RIFinish],          AT [RIEventLoc,1];
        RTemp2 ← RStatParityErr,   AT [RIEventLoc,2];
        NOP,                      AT [RIEventLoc,3];
        CALL [RSetStatus];

* Character received, store it in buffer.  If not enough room in buffer, advance to next
* IOCB unless RCmdEnd set in which case set DATALOST and wait for frame to end.
* If the current state is zero, check if SYN characther and throw away if so.

RIStoreChar: $RFStateDisp[RFState];
        DISP[.+1], T ← RSyncChar;

        GOTO [RICheckSyn], LU ← (RData) - T,                          AT [RISynLoc,0];
        GOTO [RINoCheck], T ← RRCOunt ← (RRCount) + 1,               AT [RISynLoc,1];
        GOTO [RINoCheck], T ← RRCount ← (RRCount) + 1,               AT [RISynLoc,2];
        GOTO [RICheckSyn], LU ← (RData) - T,                          AT [RISynLoc,3];

RICheckSyn: GOTO [RINoCheck, ALU#0], T ← (RRCount) ← (RRCount) + 1;
        GOTO [RINoSave];
RINoCheck: LU ← (RMaxCount) - T;
        GOTO [RIBufSpace, ALU>=0], LU ← (RCommand) AND (RCmdEnd);
RIBufFull:      GOTO [RIMustEnd, ALU#0], RRCount ← (RRCount) - 1;
        NOP;
        CALL [RDoneIOCB];
        GOTO [RIStoreChar];
RIMustEnd:      CALL [RSetStatus], RTemp2 ← RStatLost;
        GOTO [RIEndFrame];

* Room in buffer.  Store character in buffer.  This is slightly complicated due to the
* fact that we are storing a byte rather than a word.  ROffset+RRCount gives the byte
* offset into the buffer.

RIBufSpace: ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        GOTO [RIStoreEven, R EVEN], T ← RSH[ROffSet,1];
        PFetch1[RBufBase,RTemp2];
        T ← RHMASK[RData], TASK;
        RTemp2 ← (LHMASK[RTemp2]) OR T;
        GOTO [RIDoStore];
RIStoreEven: T ← LSH[RData,10];
        RTemp2 ← T;
RIDoStore:      T ← RSH[ROffSet,1], TASK;
        PStore1[RBufBase,RTemp2];
RIDoBCC:        CALL [RDoBCC];

* Now dispatch on frame state
* States are:
* 0 => just inputted a character, check if special case
* 1 => just inputting first part of BCC, increment state
* 2 => just inputted second half of BCC, check it and end frame
* 3 => counting down to end of frame, if RFState<0 end the frame

        $RFStateDisp[RFState];
        DISP[.+1];

* State = 0.  Normal state, check if special type of character.
* Special types are:
* 0 => not special type.  Do nothing.
* 1 => BCC starts accumulating AFTER this character.  Set RBCC ← 0
* 2 => BCC received AFTER this character.  Increment FState to receiving BCC state.
* 3 => Frame ends in n characters.  Set FState to countdown to end of frame.

        $RICharDisp[RTemp2],                                        AT [RIFrameLoc,0];
        DISP [.+1], RTemp2 ← (LDF[RTemp2,16,2])-1;
        GOTO [RIFinish],                                           AT [RISpecLoc,0];
        GOTO [RIFinish], RBCC ← 0C,                                AT [RISpecLoc,1];
        GOTO [RIFinish], RFState ← (RFState) + (RFStateIncr),      AT [RISpecLoc,2];
        T ← LSH[RTemp2,14],                                         AT [RISpecLoc,3];
        RFState ← (RFState) + (RFStateIncr3);
        GOTO [RICheckEOF], RFState ← (RFState) OR (T);

* State = 1.  Inputting first half of BCC.  Increment FState

        GOTO [RIFinish], RFState ← (RFState) + (RFStateIncr),      AT [RIFrameLoc,1];

* State = 2.  Got second half of BCC.  Check that accumulated BCC is equal to zero
* and set BCC error if not.  Then end the frame.

        LU ← RBCC,                                                 AT [RIFrameLoc,2];
        GOTO [RIBCCOk, ALU=0];
        NOP;
        CALL [RSetStatus], RTemp2 ← RStatBCCErr;
RIBCCOk:        GOTO [RIEndFrame], RFState ← (RFState) - (RFStateIncr2);
```

```
* State=3.  Ending frame after n characters.  Decrement top part of RFState checking it
* first to see if has gone negative.  If it has, end of frame has occurred, so reset
* FState and end the frame.

RICheckEOF: SKIPON [R<0], RFState ← (RFState) - (RFCountIncr),         AT [RIFrameLoc,3];
        GOTO [RIFinish];
        RFState ← (RFState) - (RFStateIncr3);
RIEndFrame: RFState ← (RFState) AND NOT (RFStateCountBits);
        CALL [RDoneIOCB], RState ← (RState) AND NOT (RIBStateMask);
        SKIPON [ALU=0], LU ← (RCommand) AND (RCmdStart);
        DBLGOTO [RIEndFrame,RIFinish, ALU=0];
        GOTO [RINoSave];

* Save registers in IOCB
* Save registers in CSB
* Check for more work

RIFinish:       CALL [RSaveIOCB];
RINoSave:       CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [RChkActive], RState ← (RState) AND NOT (RIActive);

* No input IOCB, mark Data Lost

RIDataLost: GOTO [RINoSave], RPNew ← (RPNew) OR (RFromDCELost);
```

```
        SET TASK[RFTask];
        ONPAGE [RFPage];

* Output character has been sent.  Load up registers and check if sending BCC.  If not,
* look at IOCB buffer for a character.

ROFrameStart: CALL [RGetPtrIOCB], T ← (RPtrCSB) + (RIndexIOCBOut);
        CALL [RGetPtrs], UseCTask;

* Need a character.  Dispatch on state
* States are:
* 0 => Line idle.  Send syncs
* 1 => Sending syncs.
* 2 => Just sent a character.  Get next character from IOCB.
* 3 => Just sent character before BCC. Send first half of BCC.
* 4 => Just sent first half of BCC. Send second half of BCC.
* 5 => Just sent last half of BCC. Idle transmitter.
* 6 => Just idled transmitter.  Signal IOCB complete.

RODispatch: $RFStateDisp[RFState];
        DISP[.+1], T ← RRCount ← (RRCount) - 1;

* Idle line.  Send first sync and load sync counter

        T ← RSyncChar,                                          AT [ROFrameLoc,0];
        RData ← T;
        T ← ($RSyncCount[RDataMask]) - 1;
        ROSyncCount ← T;
        RFState ← (RFState) + (RFStateIncr);

* Sending syncs.  Check if enough have been sent, if so start sending characters

        SKIPON [R<0], ROSyncCount ← (ROSyncCount) - 1,          AT [ROFrameLoc,1];
        GOTO [RONoSave];
        RFState ← (RFState) + (RFStateIncr);
        GOTO [ROCheckCount], LU ← RRCount;

* A character has been sent, see if any more characters in buffer.
* If nothing in buffer, check EndFrame bit and if set, end the frame.

ROCheckCount: GOTO [ROGotChar, ALU>=0], LU ← (RCommand) AND (RCmdEnd), AT [ROFrameLoc,2];
        GOTO [ROEndFrame,ALU#0], RRCount ← (RRCount) + 1;
RONextIOCB: NOP;
        CALL [RDoneIOCB];
        GOTO [RODispatch];

* Got character, send it.  Again, things are slightly complicated by the fact that we
* are sending a byte rather than a word.

ROGotChar:      T ← (RMaxCount) - T;
        ROffSet ← (ROffSet) + T;
        ROffSet ← (ROffSet) - 1;
        T ← RSH[ROffSet,1], TASK;
        PFetch1[RBufBase, RData];
        GOTO [ROSendOdd, R ODD], LU ← ROffSet;
ROSendEven: GOTO [RODoBCC], RData ← RSH[RData,10];
ROSendOdd:      GOTO [RODoBCC], RData ← RHMASK[RData];

* End of frame, since we got here, we must want to end frame, so set state to idling
* transmitter.

ROEndFrame: RFState ← (RFState) + (RFStateIncr4);
        GOTO [RONoSave], RData ← 100000C;

* Just sent character before BCC, send first BCC character and increment state

ROSendBCC:      T ← RHMASK[RBCC],                                         AT [ROFrameLoc,3];
        RData ← T;
ROIncrFrame: GOTO [RONoSave], RFState ← (RFState) + (RFStateIncr);

* Just sent first half of BCC, send the second half.

        GOTO [ROSendBCC], RBCC ← RSH[RBCC,10],                  AT [ROFrameLoc,4];

* Just sent second half of BCC, look for more characters

ROIdleXmtr: RFState ← (RFState) - (RFStateIncr3),               AT [ROFrameLoc,5];
        GOTO [ROCheckCount], LU ← RRCount;

* Just idled transmitter, mark IOCB complete and advance to next IOCB

        GOTO [RONextIOCB], RFState ← (RFState) - (RFStateIncr6),       AT [ROFrameLoc,6];

* Calculate BCC and check for special characters
* Save IOCB registers
* Save CSB registers
* Check for more work

RODoBCC:        CALL [RDoBCC];
        $RICharDisp[RTemp2];
        DISP [.+1];
        GOTO [ROFinish],                                        AT [ROSpecLoc,0];
        GOTO [ROFinish], RBCC ← 0C,                             AT [ROSpecLoc,1];
        GOTO [ROFinish], RFState ← (RFState) + (RFStateIncr),   AT [ROSpecLoc,2];
ROFinish:       CALL [RSaveIOCB],                                         AT [ROSpecLoc,3];
RONoSave:       CALL [RSaveCSB], T ← (RPtrCSB) + (RIndexIOCBOut);
        GOTO [RChkActive], RState ← (RState) AND NOT (ROActive);
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Finish off IOCB buffer

RDoneIOCB:      UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
RDoneIOCB1:LU ← (RComp) AND (RCmpStatus);
        SKIPON [ALU#0];
        RComp ← (RComp) + (RStatOk);
        CALL [RSaveIOCB], RComp ← (RComp) OR (RCmpProc);
        LoadPage[RBPage];
        CALL [RNextIOCB], T ← (RPtrIOCB) + (RIndexNext);
        T ← (RCommand) AND (OR@[RCmdWakeAll!,RCmdWakeErr!]C);
        LU ← (RComp) AND T;
        GOTO [RGetIOCB, ALU=0], T ← (RPtrCSB) + (RIndexMask);
        LoadPage[RBPage];
        CALLP [RPost], PFetch1[RBasePage0, RTemp2];
        GOTO [RGetIOCB];

* Get buffer pointers.

RGetPtrIOCB: RETURN, PFetch4[RBasePage0,RPtrIOCB];
RGetPtrs:       T ← APC&APCTask;
        RSave ← T;
RGetIOCB:       T ← RPtrIOCB;
        GOTO [RNoIOCB, ALU=0];
        CALL [RRTask], PFetch4[RBasePage0,RBufBase];
        T ← (RPtrIOCB) + (RIndexCount);
        CALL [RFixBaseReg], PFetch4[RBasePage0,RRCount];
        GOTO [RSubReturn];

* What to do if we don't have an IOCB!
* On input, signal data lost (NOT IMPLEMENTED YET)
* On output, idle line

RNoIOCB:        DBLGOTO[RINoIOCB, RONoIOCB, R ODD], LU ← RFState;
RINoIOCB:       GOTO [RINoSave];
RoNoIOCB:       GOTO [RONoSave], RData ← 100000C;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Accumulate BCC
* Load value from special character table

RDoBCC: UseCTask;
        T ← APC&APCTask, TASK;
        RSave ← T;
        T ← (RPtrCSB) + (RIndexBCCTable);
        CALL [RFixBaseReg], PFetch2[RBasePage0, RBufBase];
        T ← RData;
        T ← (RHMASK[RBCC]) XOR (T), TASK;
        PFetch1[RBufBase,RTemp2];
        T ← (RTemp2) ;
        RBCC ← (RSH[RBCC,10]) XOR T;

* Check if this character requires special handling
* Types of characters
* 0= normal
* 1=start BCC
* 2=end BCC (ie generate or check BCC).  Implies end of block on input
* 3=end of block

RCheckSpec: T ← (RData) + (400C);
        CALL [RRTask], PFetch1 [RBufBase, RTemp2];
RSubReturn: APC&APCTask ← RSave;
RRTask: RETURN, LU ← RPtrIOCB;
```

```
        SET TASK[RFTask];
        ONPAGE[RFPage];

* Fix up base register

RFixBaseReg: T ← (LSH[RBufBaseLo,10]) + 1;
        RETURN, RBufBaseLo ← (RBufBaseLo) OR (T);


* Set completion status

RSetStatus: LU ← (RComp) AND (RCmpStatus);
        GOTO [RRTask, ALU#0], T ← (RTemp2) OR (RCmpError);
        RComp ← (RComp) OR (T);


* Save IOCB ptrs

RSaveIOCB:      T ← (RPtrIOCB) + (RIndexCount);
        RETURN, PStore4[RBasePage0, RRCount];


* Save CSB ptrs

RSaveCSB:       PFetch1[RBasePage0, RPtrIOCB];
        LU ← RPtrIOCB;
        RETURN, PStore4[RBasePage0, RPtrIOCB];

        ON PAGE [RBPage];

* Point to next IOCB

RNextIOCB:      PFetch1[RBasePage0, RNext];
        DBLGOTO[.+1,.+2, R ODD], LU ← RFState;
        GOTO [.+2], T ← (RPtrCSB) + (RIndexIOCBIn);
        GOTO [.+1], T ← (RPtrCSB) + (RIndexIOCBOut);
        LU ← RNext;
        PStore1[RBasePage0,RNext];
        T ← RNext;
        RETURN, RPtrIOCB ← T;


* Post Complete

RPost:  RStackSave ← pNWW;
        T ← (GetRSpec[103]) XOR (377C); * read stkp
        STKP ← RStackSave, RStackSave ← T, NoRegILockOK;
        T ← RTemp2;
        Stack ← (Stack) OR (T);
        RTemp2 ← pRSImage;
        STKP ← RTemp2;
        T ← Stack ← (Stack) OR (10C);
        RS232 ← T;
        RETURN, STKP ← RStackSave;


        END;
```

```
*       Last modified by BRD on June 1, 1979  12:13 PM
*       Version 3.0

insert[RS232Byte];
insert[RS232Test];

END;
```

```
* Last modified by Danielson on August 28, 1979  1:06 PM, fix conflicts caused by moving CSB to GlobalDefs
* modified by Chang on August 22, 1979  6:04 PM, move CSB to Globaldefs
*  modified by Chang on August 2, 1979  1:12 PM, change CSB assignment
*  modified by BRD on June 3, 1979  4:40 PM

TITLE [RS232Defs];              * Definitions for RS232C microcode


Set[RBPage,2];
Set[RFPage,1];
Set[RFTask,4];
Set[RBTask,5];

* Register used to notify at correct task

        SET TASK[0];

RV[R0,           0];    * Register 0

* Registers (Timer task) must match those used by Kernel

        SET TASK[16];

RV[RXNotify,     52];   * Input/Output/Poller Notify reg
RV[RFNotify,     51];   * Frame Notify reg

* Registers (RS232 Frame task)

        SET TASK[RFTask];

RV[RTemp0,           0]; * Unused
RV[RTemp2,           1]; * Temporary reg
RV[RPNew,            2]; * New FromDCE bits
RV[RSave,            3]; * Subroutine return save

* The following 12 registers are loaded using three PFetch4s.  They should be quad word
* aligned

RV[RBufBase,         4]; * Base of IOCB buffer (even reg)
RV[RBufBaseLo,       5]; * Base of IOCB buffer (odd reg)
RV[RNext,            6]; * IOCB next pointer
RV[ROffset,          7]; * IOCB offset field

RV[RRCount,         10]; * IOCB count field
RV[RMaxCount,       11]; * IOCB max count field
RV[RCommand,        12]; * IOCB command field
RV[RComp,           13]; * IOCB completion IOCB field

RV[RPtrIOCB,        14]; * Ptr to start of IOCB
RV[RData,           15]; * Data from bit task
RV[RFState,         16]; * Frame state
RV[RBCC,            17]; * Frame BCC


* Registers (RS232 Poller)

RV[RPToDCE,         10]; * Command/ToDCE from CSB
RV[RPFromDCE,       11]; * FromDCE from CSB

* Registers (RS232 Bit task)

        SET TASK[RBTask];

RV[RState,          20]; * State:
RV[RDataMask,       21]; * Mask containing character length info
RV[RIData,          22]; * Input data register
RV[ROData,          23]; * Output data register

RV[RStackSave,      24]; * Register to save current stack pointer
RV[RBTemp1,         25]; * Bit task temporary
RV[RFrameTimer1,    26]; * Frame wakeup timer
RV[RPtrCSB,         27]; * Pointer to CSB

RV[RIFullBitLo,     30]; * Input bit timer
RV[RIFullBitHi,     31];
RV[RIHalfBitLo,     32]; * Input half bit timer (start bit detection)
RV[RIHalfBitHi,     33];
RV[ROFullBitLo,     34]; * Output bit timer
RV[ROFullBitHi,     35];

RV[RISyncTimer,     30]; * Input timer register
RV[ROSyncTimer,     31]; * Output timer register
RV[ROSyncCount,     32]; * Output sync counter
RV[RSyncChar,       33]; * SYN character

RV[RI1Count,        32]; * Input 1 counter
RV[RO1Count,        33]; * Output 1 counter

RV[RBasePage0,      36]; * Base for page 0
RV[RBasePage0Lo,    37];
```

```
* IOCB

* Fields in IOCB Command

MC[RCmdWakeAll,100000]; * WakeUpAlways
MC[RCmdWakeErr,040000]; * WakeUpOnError
MC[RCmdStopErr,020000]; * StopOnError
MC[RCmdStart,  010000]; * StartOfFrame
MC[RCmdEnd,    004000]; * EndOfFrame
MC[RCmdReserve,002776]; * LeftOverBits
MC[RCmdCommand,000001]; * Command field

* Fields in IOCB Completion

MC[RCmpProc,    100000]; * Processed
MC[RCmpError,   040000]; * Error
MC[RCmpEnd,     020000]; * EndOfFrame
MC[RCmpReserve,017760]; * LeftOverBits
MC[RCmpStatus, 000017]; * Completion status

* Status field in IOCB Completion

MC[RStatUnused,      0]; * Not used
MC[RStatOk,          1]; * Successful
MC[RStatLost,        2]; * DataLost
MC[RStatBreak,       3]; * Break detected
MC[RStatTOut,        4]; * TimeOut
MC[RStatBCCErr,      5]; * Checksum error
MC[RStatParityErr,   6]; * Parity Error
MC[RStatFrameErr,    7]; * FramingError
MC[RStatBadChar,    10]; * InvalidChar
MC[RStatBadFrame,   11]; * InvalidFrame
MC[RStatAbort,      12]; * IOCB aborted
MC[RStatAbort2,     13]; * IOCB aborted by TransferNow
MC[RStatDisater,    14]; * Disaster
```

```
* RState Registers fields

* Input bit states

MC[RIBStateMask,      000360];     * Input bit state bits + input parity bit
MC[RIBStateIncr,      000020];     * Increment for input bit state

* Output bit states

MC[ROBStateMask,      000017];     * Output bit state bits + output parity bit
MC[ROBStateIncr,      000002];     * Increment for output bit state
MC[ROBStateIncr2,     000004];
MC[ROBStateIncr3,     000006];
MC[ROBStateIncr4,     000010];
MC[ROBStateIncr5,     000012];

* RState Misc

MC[RXActive,          100000];     * Someone active (must be minus bit)
MC[RIActive,          040000];     * Input active
MC[ROActive,          020000];     * Output active
MC[RPActive,          010000];     * Poller active
MC[RBreakInProgress,000400];       * Break in progress
MC[RActiveMask,       170000];     * Activity bits

MC[RIParityMask,      000200];     * Input parity mask
MC[ROParityMask,      000001];     * Output parity mask

SET[RIParityBit,          10];     * Input parity bit
SET[ROParityBit,          17];     * Ouput parity bit

SET[RStateResetBit, 110004];       * State of RState after Reset command
```

```
*FState bits

* Note: Upper byte is cleared at end of frame

MC[RFCountIncr,      010000];    * Increment value to countdown
MC[RFStateCountBits,170000];     * Bits used in EOF countdown
MC[RFStateMask,      000016];    * Fstate bits
MC[RFStateInput,     000001];    * Receiver active (must be ODD bit)

MC[RFStateIncr,      000002];    * FState increments
MC[RFStateIncr2,     000004];
MC[RFStateIncr3,     000006];
MC[RFStateIncr4,     000010];
MC[RFStateIncr5,     000012];
MC[RFStateIncr6,     000014];
MC[RFStateIncr7,     000016];
```

```
* RIData Events

MC[RIDataGoodChar,  000000];    * Got a good character (async, byte sync, bit sync)
MC[RIDataBreak,     000400];    * Break detected (async)
MC[RIDataAbrt,      000400];    * Got an abrt (bit sync)
MC[RIDataParityErr, 001000];    * Parity error (async)
MC[RIDataFlag,      001000];    * Got a flag (bit sync)
MC[RIDataFrameErr,  001400];    * Framing error (async)
MC[RIDataIdle,      001400];    * Idle line (bit sync)

* RDataMaskBits

MC[RDataSlow,       000001];    * Slow speed, less than 1200 baud
MC[RDataSizeMask,   000360];    * Bits for data size in bits
```

```
* Indices into CSB and IOCB

MC[RIndexStartStop,      0];     * Index to start/stop status
MC[RIndexMask,           1];     * Index to naked notify mask bits
MC[RIndexToDCE,          2];     * Index to ToDCE bits/Command
MC[RIndexFromDCE,        3];     * Index to FromDCE bits
MC[RIndexIOCBOut,        4];     * Index to Output IOCB pointer in CSB
MC[RIndexCharOut,        5];     * Index to Output character buffer
MC[RIndexFStateOut,      6];     * Index to Output FState
MC[RIndexBCCOut,         7];     * Index to Output BCC
MC[RIndexIOCBIn,        10];     * Index to Input IOCB pointer in CSB
MC[RIndexCharIn,        11];     * Index to Input character buffer
MC[RIndexFStateIn,      12];     * Index to Input FState
MC[RIndexBCCIn,         13];     * Index to Input BCC
MC[RIndexBCCTable,      14];     * Index to BCC table pointer
MC[RIndexParm,          16];     * Index to Parameter block pointer

MC[RIndexBufPtr,         0];     * Index to Buffer pointer in IOCB
MC[RIndexNext,           2];     * Index to Next IOCB field in IOCB
MC[RIndexOffset,         3];     * Index to Offset field in IOCB
MC[RIndexCount,          4];     * Index to Count field in IOCB
MC[RIndexMaxCount,       5];     * Index to MaxCount field in IOCB
MC[RIndexCommand,        6];     * Index to Command field in IOCB
MC[RIndexComp,           7];     * Index to Completion field in IOCB
```

```
* TIMER CONSTANTS

SET[RTimerValue,377];     * Negative timer value for BSC timers
SET[RITimerSlotLo,0];     * Input timer slot (low)
SET[RITimerSlotHi,1];     * Input timer slot (hi)
SET[ROTimerSlotLo,2];     * Output timer slot (lo)
SET[ROTimerSlotHi,3];     * Output timer slot (hi)
SET[RFTimerSlotLo,15];    * Frame timer
SET[RTimerIdle,4];                * Idle timer state
SET[ROTimerSync,13];              * Output synchronous
SET[RTimerCount,14];              * Count down timer
SET[RITimerSync,17];              * Input synchronous
SET[RTimerSimple,5];              * Simple timer
SET[RFTimerValue,3];              * 3*4.48 usec per frame dispatch

SET[ROShortTimerLO,ADD[LSHIFT[RTimerCount,14],LSHIFT[1,4],ROTimerSlotLo]];
SET[RFShortTimerLo,ADD[LSHIFT[RTimerSimple,14],LSHIFT[RFTimerValue,4],RFTimerSlotLo]];
SET[RIIdleTimerLo,ADD[LSHIFT[RTimerIdle,14],LSHIFT[RTimerValue,4],RITimerSlotLo]];
SET[ROIdleTimerLo,ADD[LSHIFT[RTimerIdle,14],LSHIFT[RTimerValue,4],ROTimerSlotLo]];
SET[RFIdleTimerLo,ADD[LSHIFT[RTimerIdle,14],LSHIFT[RTimerValue,4],RFTimerSlotLo]];
SET[RISyncTimerLo,ADD[LSHIFT[RITimerSync,14],LSHIFT[RTimerValue,4],RITimerSlotLo]];
SET[ROSyncTimerLo,ADD[LSHIFT[ROTimerSync,14],LSHIFT[RTimerValue,4],ROTimerSlotLo]];
```

```
* MISC

* SET[RS232CSBLoc,177600];           * CSB location, moved to Globaldefs
MC[RI232Data,000200];               * Bit to mask off data bit on input
MC[RMaskParity,160000];             * Mask for parity type bits
SET[RBCCRemainder,170270];          * Bit sync checksum remainder

* To DCE bits

MC[RToDCEDTR,000040];                * Data terminal ready
MC[RToDCERTS,000200];                * Request to send
MC[RToDCEMask,000240];               * DTR, RTS

* From DCE bits

MC[RFromDCERing,000001];                     * Ring Indicator
MC[RFromDCEunused1,000002];                  * Unused
MC[RFromDCEunused2,000004];                  * Unused
MC[RFromDCEunused3,000010];                  * Unused
MC[RFromDCECD, 000020];              * Carrier detect
MC[RFromDCEDSR,000040];              * Data Set Ready
MC[RFRomDCECTS,000100];              * Clear to Send
MC[RFromDCEBreak,000200];                    * Break Detected
MC[RFromDCELost,000400];                     * Data Lost (no IOCB)
MC[RFromDCEMask,000161];                     * RI, CD, DSR, CTS
MC[RLatchedMask,000601];                     * Break, RI, DataLost
MC[RNotLatchedMask,000176];                  * Not LatchedMask

* Following two definitons must match those in kernel and mesa microcode.

MC[pRSImage,342];                            * Image of RS232 in timer/kernel
* MC[pNWW,25];                       * Interrupt register
```

```
* RS232 AT Locations

* The following four linkages are set up via a RS232Link SIO instruction

SET[RTWakeLoc, ADD[LSHIFT[RBPage,10], 010]];     * Receiver entry point
SET[ROWakeLoc, ADD[LSHIFT[RBPage,10], 011]];     * Transmitter entry point
SET[RPWakeLoc, ADD[LSHIFT[RBPage,10], 012]];     * Poller entry point
SET[RFWakeLoc, ADD[LSHIFT[RFPage,10], 000]];     * Frame wakeup point

SET[RS232StartLoc, ADD[LSHIFT[RFPage,10], 001]];          * RS232 Start Location
SET[RS232StartLoc1, ADD[LSHIFT[RFPage,10], 002]];         * RS232 Start Location1

SET[RIBitLoc, ADD[LSHIFT[RBPage,10],000]];       * Disp location for input bit state
SET[RIJustLoc,ADD[LSHIFT[RBPage,10],020]];       * Disp location for justifying char
SET[RIFlagLoc,ADD[LSHIFT[RBPage,10],020]];       * Disp location for input flag states
SET[RIParLoc, ADD[LSHIFT[RBPage,10],040]];       * Disp location for input parity
SET[RIIdleLoc,ADD[LSHIFT[RBPage,10],040]];       * Disp location for input idle states

SET[ROBitLoc, ADD[LSHIFT[RBPage,10],060]];       * Disp location for output bit state
SET[ROParLoc, ADD[LSHIFT[RBPage,10],360]];       * Disp location for output parity

SET[RIFrameLoc, ADD[LSHIFT[RFPage,10],20]];      * Disp location for input frame state
SET[ROFrameLoc, ADD[LSHIFT[RFPage,10], 40]];     * Disp location for output frame state
SET[RIEventLoc, ADD[LSHIFT[RFPage,10],60]];      * Disp location for input frame event
SET[RISpecLoc, ADD[LSHIFT[RFPage,10],100]];      * Disp location for input spec chars
SET[ROSpecLoc, ADD[LSHIFT[RFPage,10], 120]];     * Disp location for output spec chars
SET[RPCmdLoc, ADD[LSHIFT[RFPage,10], 140]];      * Disp locations for CSB commands
SET[RISynLoc, ADD[LSHIFT[RFPage,10], 160]];      * Disp locations for input SYN checking
```

```
* Dispatch macros

M@[$RIBitDisp,    DISPATCH[#1,11, 3]];          * Dispatch on input bit state
M@[$ROBitDisp,    DISPATCH[#1,14, 3]];          * Dispatch on output bit state
M@[$RIEventDisp, DISPATCH[#1, 6, 2]];           * Dispatch on input event
M@[$RFStateDisp, DISPATCH[#1,14, 3]];           * Dispatch on frame state
M@[$RICharDisp,   DISPATCH[#1, 0, 2]];          * Dispatch on spec char type
M@[$RIJustDisp,   DISPATCH[#1, 3, 2]];          * Dispatch on character length
M@[$RParityDisp, DISPATCH[#1, 0, 3]];           * Dispatch on parity type
M@[$RPCmdDisp,    DISPATCH[#1, 0, 4]];          * Dispatch on poller command
M@[$RStopBits,    LDF[#1, 6, 2]];               * Load stop bit count
M@[$RSyncCount,   LDF[#1,14, 3]];               * Load sync count

* Other macros

M@[SKIPON, GOTO[.+2,#1]];                       * Skip Macro

* M@[$RSetDispLo,  AND@[ADD[LSHIFT[#1,14],#2],377]C];    * Set dispatch value from task,addr
* M@[$RSetDispHi,  AND@[ADD[LSHIFT[#1,14],#2],177400]C];

        END;
```

```
Insert[D0Lang];
NoMidasInit;LangVersion;MultDIB;

        Title[RS232Occupied];

* Generated by BRD on June 25, 1979  11:54 AM

* Locations reserved on page 0

        IMRESERVE[0, 0, 400];

* Locations reserved on page 1

* Locations reserved on page 2

        IMRESERVE[2,100,210];
        IMRESERVE[2,374,4];

* Locations reserved on page 3

        IMRESERVE[3, 0, 400];

* Locations reserved on page 4

        IMRESERVE[4, 0, 400];

* Locations reserved on page 5

        IMRESERVE[5, 0, 400];

* Locations reserved on page 6

        IMRESERVE[6, 0, 400];

* Locations reserved on page 7

        IMRESERVE[7, 0, 400];

* Locations reserved on page 10B

        IMRESERVE[10, 0, 400];

* Locations reserved on page 11B

        IMRESERVE[11, 0, 400];

* Locations reserved on page 12B

        IMRESERVE[12, 0, 400];

* Locations reserved on page 13B

        IMRESERVE[13, 0, 400];

* Locations reserved on page 14B

        IMRESERVE[14, 0, 400];

* Locations reserved on page 15B

        IMRESERVE[15, 0, 400];

* Locations reserved on page 16B

        IMRESERVE[16, 0, 400];

* Locations reserved on page 17B

        IMRESERVE[17, 0, 400];

        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];

        TITLE[RS232SIO];

* Last modified by Chang on June 26, 1979  11:02 AM; move to Globaldefs
*  modified by Johnsson on June 13, 1979  3:03 PM; new registers
* modified by BRD on June 3, 1979  4:09 PM
*       added RS232 SIO address constants

* RS232 SIO instructions


* M@[$RSetDispLo,   AND@[ADD[LSHIFT[#1,14],#2],377]C];
* M@[$RSetDispHi,   AND@[ADD[LSHIFT[#1,14],#2],177400]C];

Set[RBPage,2];
Set[RFPage,1];
Set[RXDispatchLoc,ADD[LSHIFT[RBPage,10],010]];
Set[RFDispatchLoc,ADD[LSHIFT[RFPage,10],000]];



        SET TASK [16];
        ONPAGE [EEPage];

* The following two definitions must match those in Timer code.

* RV[RFNotify,46];        * Register containing frame notify values
* RV[RXNotify,47];        * Register containing bit notify values

* RS232 SIO noop (00)

RSIORet:        RETURN,                                                 AT [RS232SIOLoc,0];

* RS232 SIO stop (01)
* Set dispatch address to be RS232SIOLoc+4

        RXNotify ← $RSetDispLo[RBTask,ADD[RS232SIOLoc,4]],         AT [RS232SIOLoc,1];
        T ← RXNotify ← (RXNotify) OR ($RSetDispHi[RBTask,ADD[RS232SIOLoc,4]]);
        RETURN, RFNotify ← T;

* RS232 SIO start (02)

        RXNotify ← $RSetDispHi[RBTask,RXDispatchLoc],              AT [RS232SIOLoc,2];
        RXNOtify ← (RXNotify) OR ($RSetDispLo[RBTask,RXDispatchLoc]);
        RFNotify ← $RSetDispHi[RFTask,RFDispatchLoc];
        RETURN, RFNotify ← (RFNotify) OR ($RSetDispLo[RFTask,RFDispatchLoc]);

* RS232 SIO unused (03)

*       RETURN,                                                 AT [RS232SIOLoc,3];

* Following four RETURNs used to turn off RS232 notifies

RS232Ret:               RETURN,                                     AT [RS232SIOLoc,4];
        RETURN,                                                 AT [RS232SIOLoc,5];
        RETURN,                                                 AT [RS232SIOLoc,6];
*       RETURN,                                                 AT [RS232SIOLoc,7];

        end[RS232SIO];
```

```
* Test program for RS232C Microcode Loopback test
* Version 3.0
* Last modified by BRD on April 30, 1979  6:22 PM

SET[TestPage,4];

        SET TASK[0];
        ON PAGE[TestPage];

* Restart location.

RV[R0, 0];                                        * Temp
RV[R1, 1];                                        * Temp
RV[R2, 2];                                        * Temp
RV[RBase0,4];                           * Page 0 base reg
RV[RBase0Lo,5];                         * Page 0 base reg
RV[RInBuf,6];                           * Input buf base reg
RV[RInBufLo,7];                         * Input buf base reg
RV[ROutBuf,10];                         * Output buf base reg
RV[ROutBufLo,11];                                 * Output buf base reg
RV[RInBase,12];                         * Input IOCB pointer
RV[RInByte,13];                         * Input character
RV[ROutBase,14];                                  * Output IOCB pointer
RV[ROutByte,15];                                  * Output character
RV[RCnt,16];                            * Count from IOCB
RV[ROff,17];                            * OffSet from IOCB
RV[RCSBBase,20];                                  * Pointer to CSB

SET[RStartLoc,ADD[LSHIFT[TestPage,10],10]];
Set[RXDispatchLoc,ADD[LSHIFT[RBPage,10],010]];
Set[RFDispatchLoc,ADD[LSHIFT[RFPage,10],000]];
Set[RS232SIOLoc,ADD[LSHIFT[TestPage,10],000]];
Set[RSLoc,ADD[RS232SIOLoc,1]];
MC[tmr38conreg,324]; *RM 324 holds 38usec timer restart constant

RS:     T ← R0;                         * Notify task 0 at Start
        R0 ← AND@[0377, RStartLoc]C;
        R0 ← (R0) OR (AND@[007400, RStartLoc]C);
        APC&APCTask ← R0;
        RETURN, R0 ← T;                 * Restore R0
RStart: ClearMpanel, AT[RStartLoc];
        RBase0 ← 0C;
        RBase0Lo ← 0C;

* Clear out memory from 1000 to 2000

        R1 ← 0C;
        T ← R2 ← 1000C;
RDoClear:       PStore1[RBase0, R1];
        T ← R2 ← (R2) + 1, CALL[RDoTask];
        LU ← (R2) - (2000C);
        GOTO [RDoClear, ALU<0];

* Start transmitter by faking SIO
        RCSBBase ← AND@[177100,377]C;   * CSB pointer
        RCSBBase ← (RCSBBase) OR (AND@[177100,177400]C);
RSetThings:BREAKPOINT, NOP;
        CALL [RSIOStart];                         * Fake SIO start
        CALL [RFBegin];                 * Initialize frame dispatch code
        CALL [RSet38Timer];                       * Start 38 usec timer
        CALL [RDoReset];                          * Issue a reset command
        CALL [RDoStopTransmitter];                * Issue a StopTransmitter command
        CALL [RDoStopReceiver]; * Issue a StopReceiver command
        CALL [RDoStartReceiver];                  * Issue a StartReceiver command
        CALL [RDoStartTransmitter];               * Issue a reset command

* Incrment MPanel and check buffers

        RInBase ← 2000C;
        RInBase ← (RInBase) + (200C);
        ROutBase ← 200C;
RLoop:  CALL [.+1];
        T ← (RInBase) + (RIndexComp);
        PFetch1[RBase0, R1];    * Check Completion
        LU ← (R1) AND (RCmpProc);
        GOTO [RGotInput, ALU#0];
        T ← (ROutBase) + (RIndexComp);
        PFetch1[RBase0, R1];    * Check Completion
        LU ← (R1) AND (RCmpProc);
        GOTO [RGotOutput, ALU#0];
        RETURN;
RGotOutput: R1 ← (R1) AND NOT (RCmpProc);
        PStore1[RBase0, R1];
        T ← (ROutBase) + (RIndexMaxCount);
        PFetch1[RBase0, R1];
        T ← (ROutBase) + (RIndexCount);
        GOTO [RLoop], PStore1[RBase0, R1];
RGotInput:      IncMPanel;
        R1 ← (R1) AND NOT (RCmpProc);
        PStore1[RBase0, R1];
        T ← (RInBase) + (RIndexBufPtr), TASK;
        PFetch2[RBase0, RInBuf];
        T ← (LSH[RInBufLo,10]) + 1;
        RInBufLo ← (RInBufLo) OR (T);
        T ← (ROutBase) + (RIndexBufPtr), TASK;
        PFetch2[RBase0, ROutBuf];
        T ← (LSH[ROutBufLo,10]) + 1;
        ROutBufLo ← (ROutBufLo) OR (T);
        T ← (RInBase) + (RIndexCount);
        PFetch1[RBase0, RCnt];
        T ← (RInBase) + (RIndexOffSet);
```

```
        PFetch1[RBase0, ROff];
RCompLoop:      T ← ROff;
        LU ← (RCnt) - T;
        GOTO [REndLoop, ALU=0];
        CALL [RGetInByte];
        CALL [RGetOutByte];
        T ← RInByte;
        LU ← (ROutByte) - T;
        GOTO [RCompLoop, ALU=0], ROff ← (ROff) + 1;
        BREAKPOINT;
REndLoop:       T ← (RInBase) + (RIndexCount);
        R1 ← 0C;
        PStore1[RBase0, R1];
        T ← (RInBase) + (RIndexNext);
        PFetch1[RBase0, RInBase];
        T ← (ROutBase) + (RIndexNext);
        GOTO [RLoop], PFetch1[RBase0, ROutBase];

* Subroutine to set up TPC link for frame code

RFBegin:        R1 ← $RSetDispLo[0,RS232StartLoc];
        GOTO [RFDoDisp], R1 ← (R1) OR ($RSetDispHi[0,RS232StartLoc]);

* Subroutine to fake RS232 SIO Start Instruction

RSIOStart:      R1 ← $RSetDispLo[16,RSLoc];
        R1 ← (R1) OR ($RSetDispHi[16,RSLoc]);
RFDoDisp:       APC&APCTask ← R1;
RDoTask:        RETURN;

* Subroutine to issue a poller commands

RDoReset:               GOTO [RDoACmd], R1 ← 20000C;
RDoStartReceiver:       GOTO [RDoACmd], R1 ← 30000C;
RDoStartTransmitter:    GOTO [RDoACmd], R1 ← 40000C;
RDoStopReceiver:        GOTO [RDoACmd], R1 ← 50000C;
RDoStopTransmitter:     GOTO [RDoACmd], R1 ← 60000C;
RDoACmd:        UseCTask;
        T ← APC&APCTask;
        R0 ← T;
        T ← (RCSBBase) + (RIndexToDCE);
        PFetch1[RBase0, R2];
        T ← R1;
        R2 ← (R2) OR (T);
        T ← (RCSBBase) + (RIndexToDCE);
        CALL [RWaitCmd], PStore1[RBase0, R2];
RWaitCmd:       NOP;
        NOP;
        NOP;
        NOP;
        PFetch1 [RBase0, R2];
        LU ← LHMASK[R2];
        SKIPON [ALU#0];
        APC&APCTask ← R0;
        RETURN;

* Set 38 usec timer

RSet38Timer: R0 ← (tmr38conreg);
        stkp ← R0 ;
        stack ← (50000c) ;
        stack ← (stack) or (176c) ; *simple timer,value 7,slot 16
        loadtimer[stack] ;
        RETURN;

* Get Input byte

RGetInByte: T ← RSH[ROff,1];
        PFetch1[RInBuf,RInByte];
        SKIPON [R ODD], LU ← ROff;
RGetInEven: RETURN, RInByte ← RSH[RInByte,10];
RGetInOdd:      RETURN, RInByte ← RHMASK[RInByte];

* Get Output byte

RGetOutByte: T ← RSH[ROff,1];
        PFetch1[ROutBuf,ROutByte];
        SKIPON [R ODD], LU ← ROff;
RGetOutEven: RETURN, ROutByte ← RSH[ROutByte,10];
RGetOutOdd: RETURN, ROutByte ← RHMASK[ROutByte];
```

```
* RS232 SIO code

        SET TASK [16];


* RS232 SIO noop (00)

RSIORet:        RETURN,                                                     AT [RS232SIOLoc,0];

* RS232 SIO start (01)

        RXNotify ← $RSetDispHi[RBTask,RXDispatchLoc],               AT [RS232SIOLoc,1];
        RXNOtify ← (RXNotify) OR ($RSetDispLo[RBTask,RXDispatchLoc]);
        RFNotify ← $RSetDispHi[RFTask,RFDispatchLoc];
        RETURN, RFNotify ← (RFNotify) OR ($RSetDispLo[RFTask,RFDispatchLoc]);

* RS232 SIO stop (02)
* Set dispatch address to be RS232SIOLoc+4

        RXNotify ← $RSetDispLo[RBTask,ADD[RS232SIOLoc,4]],          AT [RS232SIOLoc,2];
        T ← RXNotify ← (RXNotify) OR ($RSetDispHi[RBTask,ADD[RS232SIOLoc,4]]);
        RETURN, RFNotify ← T;

* RS232 SIO unused (03)

        RETURN,                                                     AT [RS232SIOLoc,3];

* Following four RETURNs used to turn off RS232 notifies

        RETURN,                                                     AT [RS232SIOLoc,4];
        RETURN,                                                     AT [RS232SIOLoc,5];
        RETURN,                                                     AT [RS232SIOLoc,6];
        RETURN,                                                     AT [RS232SIOLoc,7];

END;
```

```
BUILTIN[INSRT,24];
INSRT[DOLANG];
NOMIDASINIT;MULTDIB;
TITLE[Timer];

INSRT[GlobalDefs];

* Last Modified by Chang on August 20, 1979  6:49 PM, move timer's regs
*  Modified by Johnsson on July 10, 1979  7:10 PM, New Kernel faults
* Modified by Chang on June 1, 1979  6:18 PM, RXNotify = 52
* Modified by Sandman on May 8, 1979  12:16 PM
* Added Pilot high resolution timer;

* Modified by Sandman on April 6, 1979  7:36 PM
* Pinned down timer code;

* modified March 26, 1979  1:07 PM - added RS232C hooks
*       added poller dispatch code in realtime clock
*       added bit and frame dispatch code in timer dispatch table

* Modified March 22, 1979  9:35 PM by Chang for PushButton Boot
* replaced "CheckStop" and "CheckStop1" by "TimerRet"

* Based Kernel.mc by CPT  March 1, 1979

IMRESERVE[1,0,100]; * Don't use EPROM area
IMRESERVE[2,0,100]; * Don't use EPROM area

RV[temp,51];
RV[initr0,52];
RV[initr1,53];
RV[initr2,64];
RV[initr3,65];

* The following definitions MUST MATCH THOSE IN RS232C MICROCODE!!!

SetTask[TTask];
RV[EONotify, 40];         * Register containing notify value for Ethernet
RV[RFNotify,46];          * Register containing frame notify values
RV[RXNotify,47];          * Register containing bit notify values
* End of RS232 definitions

RV[RSImage,42]; *Image of RS232 hardware register

* RV[RWO,37];*temporary for ControlStore address during initial clear and read/write
RV[RWO,56];*temporary for ControlStore address during initial clear and read/write


* RV[RTCLOW,25];                  *low half of Alto RealTime clock
RV[RTCLOW,55];            *low half of Alto RealTime clock
* RV[TMR38CON,24];                *constant for 38usec timer
RV[TMR38CON,54];                  *constant for 38usec timer
RV[RTIMER,57];            *refresh timer register
RV[REFR,77];              *refresh address
RV[R377,77];              *use this register to clear R file

SET[BootStartLoc,add[lshift[TimerinitPage2,10],372]];*Push button start loc.
SET[ReadyToGoLoc,add[lshift[TimerinitPage2,10],374]];*End of PreInitialization

                OnPage[TimerinitPage2];
                SetTask[0];
BootStart:
        temp ← (1000c), at[BootStartLoc]; * set task #0
        temp ← (temp) or (116C);
        apc&apctask ← temp; * gotop Write CS Loc1
        Return;

MC[FaultLoc, 100];
Set[BeginFault, 100];
                OnPage[TimerinitPage1];
                SetTask[0];
* Write at Location 1:
*       T ← APC&APCTask, goto[FaultOccured], AT[1];
*
        initr0 ← (50000C), at[1116];
        initr0 ← (initr0) or (150c);
        initr1 ← (65000C);
        initr1 ← (initr1) or (1c);
        initr2 ← (15C);
        initr3 ← (1C);
        t ← initr2 ;
        LU ← initr0;     *T has data 2
        APC&APCTASK ← initr3;
        WRITECS0&2;
        LU ← initr1;
        APC&APCTASK ← initr3;
        WRITECS1;
        initr0 ← (45C);
        initr1 ← (1000C);
        initr1 ← (initr1) or (177C);
        initr2 ← (3C);
        initr3 ← (0C);
        t ← initr2 ;
        LU ← initr0;     *T has data 2
        APC&APCTASK ← initr3;
        WRITECS0&2;
        LU ← initr1;
        APC&APCTASK ← initr3;
        WRITECS1;
        temp ← (161000C); * notify task #16
        temp ← (temp) or (376C);
```

```
	apc&apctask ← temp;
	Return;

		SetTask[TTask];
	RTMP ← (100000C), at[1376];
CLRTIMERS:	LOADTIMER[RTMP]; *Clear out all Timers
	T ← 0C;
	RS232 ← T;
	RTMP ← (RTMP) + 1, RESETMEMERRS; *Clear any pending memory errors
	LU ← (RTMP) AND (17C);		*there are 16d timers
	REFR ← (0C), DBLGOTO[INITDONE, CLRTIMERS, ALU=0];
INITDONE:
	LoadPage[TimerPage];
	RSImage ← 0c, gotop[SetUpRef];

		OnPage[TimerPage];
		SetTask[TTask];

SET[TimerBase, ADD[LSHIFT[TimerPage, 10], 300]];
SET[TimerTable, ADD[LSHIFT[TimerPage, 10], 340]];
SET[AuxTimerTable, ADD[LSHIFT[TimerPage, 10], 360]];
SET[RefreshBase, ADD[LSHIFT[TimerPage, 10], 244]]; * share mc1 dispatch table

SetUpRef:
	LU ← TIMER;		*Set up the Refresh timer
	RTIMER ← (50000C);
	RTIMER ← (RTIMER) OR (257C);	*simple timer,value 10d,slot 17b
	LOADTIMER[RTIMER];
	call[TimerReturn], AT[TimerBase, 35];
TimerWakeup:		*Timer wakeups come here
	t ← (Dispatch[Timer,14,4]), AT[TimerBase, 36];
	DISP[Timers], AT[TimerBase, 37];

TimerReturn:
	RTMP ← (1000C); * transfer to task #0
	RTMP ← (RTMP) or (374C);
	apc&apctask ← RTMP; * goto ReadyToGo
	Return;

		OnPage[TimerinitPage2];
		SetTask[0];
ReadyToGo:
	LoadPage[InitPage], at[ReadyToGoLoc];* goto START of Initialization
	gotop[START];

		OnPage[0];
		SetTask[17];
*Page Zero stuff
*We put the instruction for BufferRefill here..
*----------------------------------------
*x377x: gotop[.], at[377]; *dummy
* following 2 words replaced by the microcodes
*	loadpage[0], goto[x377x], at[0]; *buffer refill code is on page 0
*	T ← APC&APCTask, goto[FaultOccured], AT[1];	*First, must save apc
*----------------------------------------

FaultOccured:
	RXAPC ← T, AT[BeginFault];
	T ← GETRSPEC[147], AT[101];	*ctask, ncia
	RXCTASK ← T, AT[102];
	T ← (GETRSPEC[103]) xor (377c), AT[103];	*sstkp, stkp (stkp is read complemented)
	RXSTK ← T, AT[104];
	RTMP ← 20c, AT[105]; *Set stkp to 20 in case there was a stack overflow pending
	Stkp ← RTMP, AT[106];
	T ← (GETRSPEC[107]) xnor (0c), AT[107]; *aluresult, saluf (both read complemented)
	RXALU ← T, AT[110];
	T ← GETRSPEC[157], LOADPAGE[0], AT[111]; *page, parity, bootreason
	RXPPB ← T, RESETERRORS, GOTO[FaultStart], AT[112];

		OnPage[TimerPage];

*Timer dispatch table for task 16
		SetTask[TTask];
Timers:
	REFRESH[REFR], goto[RefreshNext], AT[TimerTable,00] ;*slot 17
	ADDTOTIMER[TMR38CON], goto[RTCNext], AT[TimerTable,01]; *slot 16
	GOTO [TimerRet], APC&APCTask ← RFNotify, AT[TimerTable,02]; *slot 15 -- frame dispatch
	RETURN, AT[TimerTable,03]; *slot 14
	RETURN, AT[TimerTable,04]; *slot 13
	APC&APCTask ← EONotify, GOTO[TimerRet], AT[TimerTable,05]; *slot 12 (Ethernet slot = EOTask)
	RETURN, AT[TimerTable,06]; *slot 11
	RETURN, AT[TimerTable,07]; *slot 10
	RETURN, AT[TimerTable,10]; *slot 07
	APC&APCTask ← EONotify2, GOTO[TimerRet], AT[TimerTable,11]; *slot 06 (Ethernet slot = EOTask2)
	RETURN, AT[TimerTable,12]; *slot 05
	RETURN, AT[TimerTable,13]; *slot 04
	RXNotify ← (RXNotify) OR (RONotify), GOTO[.+2], AT[TimerTable,14]; *slot 03 (RS232 output)
	RXNotify ← (RXNotify) OR (RONotify), GOTO[.+1], AT[TimerTable,15]; *slot 02 (RS232 output)
RXDoNotify:
	APC&APCTask ← RXNotify, GOTO[RXReturn], AT[TimerTable,16]; *slot 01 (RS232 input)
	APC&APCTask ← RXNotify, GOTO[RXReturn], AT[TimerTable,17];*slot 00 (RS232 input)

AuxTimers:
*	goto[.], AT[AuxTimerTable, 00];	* Refresh Task (Used elsewhere)
*	goto[.], AT[AuxTimerTable, 01];	* Alto Real Time Clock (Used at RTCNext)
	goto[.], AT[AuxTimerTable, 02];
	goto[.], AT[AuxTimerTable, 03];
	goto[.], AT[AuxTimerTable, 04];
	goto[.], AT[AuxTimerTable, 05];
	goto[.], AT[AuxTimerTable, 06];
	goto[.], AT[AuxTimerTable, 07];
```

```
        goto[.], AT[AuxTimerTable, 10];
        goto[.], AT[AuxTimerTable, 11];   * Ethernet slot = EOTask (doesn't use it)
        goto[.], AT[AuxTimerTable, 12];
        goto[.], AT[AuxTimerTable, 13];
*       goto[.], AT[AuxTimerTable, 14];   * RS232 output (Used at RXReturn + 1)
*       goto[.], AT[AuxTimerTable, 15];   * RS232 output (Used at RXReturn)
*       goto[.], AT[AuxTimerTable, 16];   * RS232 input (Used by RTClock at RTCNext+1)
        goto[.], AT[AuxTimerTable, 17];   * RS232 input (doesn't use it)
        SETTASK[16];

*Refresh has been started.  Increment the address.
RefreshNext:
        REFR ← (REFR) + (20C), AT[RefreshBase, 0];
        ClockLo ← (ClockLo) + 1, AT[RefreshBase, 1];
        Skip[no Carry], REFR ← (REFR) and not (1000C), AT[RefreshBase, 4]; *no carries beyond bit 6d
          ClockHi ← (ClockHi) + 1, AT[RefreshBase, 3];
        ADDTOTIMER[RTIMER], AT[RefreshBase, 2];

* check for Midas halt

        lu ← ldf[FFault,1,1], AT[RefreshBase, 5];        * check for Midas
        T ← PRINTER, goto[TimerRet, alu=0], AT[RefreshBase, 10];
        RTMP ← T, AT[RefreshBase, 7];
        T ← RTMP ← (RTMP) and (10000c), AT[RefreshBase, 11];
        T ← (PRINTER) and (T), AT[RefreshBase, 12];
        skip[alu#0], AT[RefreshBase, 13];
          return, AT[RefreshBase, 20];
        goto[.], SETFAULT, AT[RefreshBase, 21];
        return, at[RefreshBase, 22];

TimerRet:       RETURN, AT[RefreshBase,6] ;
*Increment RTCLOW by 40.  The display task will check the
*sign bit, and increment MM 430 and clear the bit if it is on.
* Test if time for an RS232 poll (every 256 x 38 usec, ~ 10 msec)
RTCNext:
        LU ← LDF[RTCLOW, 3, 10], AT[AuxTimerTable, 01]; * Check if bits 3..12 are zero
        SKIP[ALU=0], RTCLOW ← (RTCLOW) + (40C), AT[AuxTimerTable, 16];
RXReturn:
        RXNotify ← (RXNotify) AND NOT (3C), return, AT[AuxTimerTable, 15];
        RXNotify ← (RXNotify) OR (RPNotify), GOTO[RXDoNotify], AT[AuxTimerTable, 14];

END;
```

```
TITLE[uiDEFS];
*last edit by Chang  August 20, 1979  6:36 PM, move Timer's regs
* edit by CPT  December 22, 1978  3:18 AM
* edit by Sandman  March 23, 1979  3:02 PM

*UIDEFS.MC -definitions for IUTFP revision I

SET[uiUTFPBASEADDR,LSHIFT[uiUTFPPAGE,10]]; *FIRST ADDRESS OF UTFP PAGE


*REGISTERS AND CONSTANTS USED BY UTFP TASK
SETTASK[uiUTFPTASK];

SET[uiREADSTATREG,1];
SET[uiDBREG,1];
SET[uiCREG,2];
SET[uiHEBUF,3];
SET[uiCXREG,4];
SET[uiHTAB,5];
SET[uiBPREG,6];
SET[uiCURSM,7];
SET[uiDBADDR,ADD[lshift[uiUTFPTASK,4],1]];

MC[BlkBkgndBit,100];

RV[uiDWA,0]; *bit map base register
RV[uiDWA1,1];
RV[uiTEMP,2]; *must be even/odd pair, see uiDCBDONE for PFETCH2
RV[uiTEMP1,3];
RV[uiMPSTATUS,4]; *bits 0-4: count.5:7: part.14-17: vs STATE
RV[uiMOUSEDELXY,5]; *BITS 0-5: XDELTA, 10-15: YDELTA
RV[uiTMSG,6]; *INCOMING PARTIAL MESSAGE
RV[uiXMSG,7]; *MESSAGE HELD FOR POSTING BY VSYNC

RV[uiLINK,10]; *DISPLAY CONTROL BLOCK WORD 0
RV[uiNWRDS,11]; *DISPLAY CONTROL BLOCK WORD 1
RV[uiDBA,12]; *DISPLAY CONTROL BLOCK WORD 2
RV[uiHEPAT,12]; *Used during initialization only
RV[uiSLC,13]; *DISPLAY CONTROL BLOCK WORD 3
RV[uiHEADDR,13]; *Used during initialization only
RV[uiQTEMP,10]; *Used for Store4 to post mouse buttons (uiVS4)
RV[uiQTEMP1,11];
RV[uiQTEMP2,12];
RV[uiQTEMP3,13];

RV[uiBUFPTR,14];
RV[uiHECNT,14]; *Used during initialization only
RV[uiCRWORD,15]; *IMAGE OF HARDWARE CONTROL REGISTER
RV[uiVSCOUNT,16]; *Count of lines per field.
RV[uiHELINK,16]; *Used during initialization only
RV[uiLINESPERFIELD,17]; *594 = 1122b lines per field
mc[lpflo,122];
mc[lpfhi,1000];

RV[uiCC0,10];    * used during initialization only
RV[uiCC1,11];    * used during initialization only
RV[uiCC2,12];    * used during initialization only
RV[uiCC3,13];    * used during initialization only

* RV[RTCLOW,25]; *Must be the same as timer's
RV[RTCLOW,55]; *Must be the same as timer's

RV[uiCX,30]; *Cursor X
RV[uiCY,31]; *Cursor Y
RV[uiCNT,32];
RV[uiBASE,34]; *BASE REGISTER PAIR
RV[uiBASE1,35];
RV[uiNBUFPTR,36];
RV[uiBUTTONS,37];


*       display constants
*COUNTS (BITS 1-9) AND PATTERNS (BITS 12-15) FOR HE RAM
mc[her0l,  203];
mc[her1l,  201];
mc[her1h,20400];
mc[her2l,  200];
mc[her3l,  101];
mc[her3h,12400];
mc[her4l,  203];
mc[her6l,  307];
mc[her5h, 5400];
mc[her6l,  106];
mc[her7l,  116];
mc[her8l,  107];
mc[her9l,  110];
mc[her9h,35400];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[UIDefs];
          title[UIInit];
*last edit by Johnsson  April 7, 1979  12:29 PM

*Initialization for IUTFP

SETTASK[uiUTFPTASK];
ONPAGE[DisplayInitPage];


displayinit:  uiHEADDR ← (ZERO), AT[DisplayInitLoc];
          OUTPUT[uiHEADDR,uiCREG]; *CLEAR THE CONTROL REGISTER

          uiHEPAT ← her0l, CALL[uiLOADHE]; *LOAD THE HORIZONTAL EVENT RAM

          uiHEPAT ← (her1h) ;
          uiHEPAT ← (uiHEPAT) or (her1l) ,CALL[uiLOADHE];

          uiHEPAT ← (her2l), CALL[uiLOADHE];

          uiHEPAT ← (her3h) ;
          uiHEPAT ← (uiHEPAT) or (her3l) ,CALL[uiLOADHE];

          uiHEPAT ← (her4l), CALL[uiLOADHE] ;

          uiHEPAT ← (her5h) ;
          uiHEPAT ← (uiHEPAT) or (her5l) ,CALL[uiLOADHE];


          uiHEPAT ← (her6l), CALL[uiLOADHE] ;

          uiHEPAT ← (her7l), CALL[uiLOADHE] ;

          uiHEPAT ← (her8l), CALL[uiLOADHE] ;

          uiHEPAT ← (her9h) ;
          uiHEPAT ← (uiHEPAT) or (her9l) ,CALL[uiLOADHE];

uiHELOADED:
          uiBASE ← zero;
          uiBASE1 ← zero ;
          uiLINESPERFIELD ← lpflo ;
          uiLINESPERFIELD ← (uiLINESPERFIELD) or (lpfhi) ;

*set keyboard words to -1 (key up)
          uiTEMP ← 177000C;
          T ← uiTEMP ← (uiTEMP) or (30C); * 177030
          uiCC0 ← (ZERO)-1;
          uiCC1 ← (ZERO)-1;
          uiCC2 ← (ZERO)-1;
          uiCC3 ← (ZERO)-1;
          PSTORE4[uiBASE,uiCC0];          * mouse,keyset,etc.
          T ← uiTEMP ← (uiTEMP)+(4c);     * 177034
          PSTORE4[uiBASE,uiCC0];          * keyboard[0:3].
          T ← uiTEMP ← (uiTEMP)+(4c);     * 177040
          PSTORE4[uiBASE,uiCC0];          * keyboard[4:7].

          uiCRWORD ← (220C); *ALLOW WAKEUPS, CDIAG←0
          OUTPUT[uiCRWORD,uiCREG]; *ALLOW WAKEUPS
          uiTEMP ← 377C;
          OUTPUT[uiTEMP,uiHTAB]; *load the HTAB counter with 377
          uiTMSG ← (ZERO);
          uiMPSTATUS ← T ← ZERO, call[uiFINHE];
          uiLINK ← T, loadpage[uiutfppage]; *First wakeup comes here
          uiBUFPTR ←T← 377C,GOTOp[uiCSDONE];

*SUBROUTINE TO LOAD THE HORIZONTAL EVENT RAM
*(ADDRESSED VIA CXREG)
uiLOADHE:        T ← LDF[uiHEPAT,1,11];
          uiHECNT ← (T);
uiHELOADLOOP: uiHECNT ← (uiHECNT)-1;
          GOTO[uiFINHE,ALU<0], usectask;
          OUTPUT[uiHEADDR, uiCXREG];
          uiHEADDR ← (uiHEADDR)+1;
          OUTPUT[uiHEPAT, uiHEBUF],goto[uiHELOADLOOP];
uiFINHE:         RETURN;

          end[uiinit];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[UIDefs];
TITLE[extended-address-UITASK];

*Task microcode for IUTFP
* Last modified by Chang, August 20, 1979  6:38 PM, move Timer's regs;
* modified by Johnsson, April 7, 1979  12:29 PM;

SETTASK[uiUTFPTASK];
ONPAGE[uiUTFPPAGE];


SET[uiPART,ADD[uiUTFPBASEADDR,20]]; *which part of the backchannel message is coming.
SET[uiSHCUR,ADD[uiUTFPBASEADDR,60]]; *cursor shift


*Subroutine to check for and gather messages.
*Call with DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
*Returns without tasking

uiCHKMSG:        DISP[uiPART0];  *Dispatch on MPSTATUS.PART

*Test for message (IOATTN)
uiPART0: GOTO[uiMSGRTN1, NOATTEN], usectask, AT[uiPART,0];
        uiMPSTATUS ← (uiMPSTATUS) OR (110400C),RETURN;  *START MSG,
*Set uiMPSTATUS.PART=1, uiMPSTATUS.COUNT=-14

uiPART1: T ← (1000C), DBLGOTO[uiINCXY,uiNOINCXY,IOATTEN], AT[uiPART,1]; *+X bit

uiPART2: T ← (177000C), DBLGOTO[uiINCXY,uiNOINCXY,IOATTEN], AT[uiPART,2];      *-X bit

uiPART3: T ← (1C), DBLGOTO[uiINCXY,uiNOINCXY,IOATTEN], AT[uiPART,3]; *+Y bit

uiPART4: T ← (177C), DBLGOTO[uiINCXY,uiNOINCXY,IOATTEN], AT[uiPART,4];  *-Y BIT

uiINCXY: uiMOUSEDELXY ← (uiMOUSEDELXY) + (T);
        uiMOUSEDELXY ← (uiMOUSEDELXY) AND NOT (400C);
uiNOINCXY: uiMPSTATUS ← (uiMPSTATUS) + (400C),GOTO[uiMOREMSG]; *Increment part by 1


uiPART5: T ← uiTMSG ← RSH[uiTMSG,1], DBLGOTO[uiMSGONE,uiMSGZERO,IOATTEN], AT[uiPART,5];

uiMSGONE: T ← uiTMSG ← (uiTMSG) OR (100000C);    *OR a 1 bit into the message.

*Increment (Negative) count.
uiMSGZERO: uiMPSTATUS ← (uiMPSTATUS) + (4000C), DBLGOTO[uiMOREMSG,uiENDMSG,R<0];

uiMOREMSG:  USECTASK;

uiMSGRTN1: RETURN;

*Post the keyboard and mouse buttons
uiENDMSG:  t← lcy[uiTMSG,1]; *bit 1 is the keyboard change bit
        uiBUTTONS ← T, goto[uiNOKBDCHANGE, alu>=0]; *save buttons

uiKBDCHANGE: t ← ldf[uiTMSG,4,10]; *keyboard data
        uiXMSG ← (lsh[uiXMSG,10]) or (t);

uiNOKBDCHANGE: USECTASK;
        uiMPSTATUS ← (uiMPSTATUS) AND NOT (177400C),RETURN; *Clear count and part.
```

```
*Do horizontal processing.  We know that the controller needs data.
xpreVS0:        DISPATCH[uiMPSTATUS,5,3], call[uiCHKMSG];
preVS0: T← 2C, call[uiCheckCursor];

uiVS0:  OUTPUT[uiNBUFPTR,uiBPREG]; *precomputed uiNBUFPTR to hardware.
        T ← (uiDBA) AND NOT (17C);
        uiDWA ← T;
*Start the first IOFETCH
uiDWT:  IOFETCH16[uiBASE,uiDBADDR];

*Calculate the read buffer pointer, the count, and next line's DBA
*in the shadow of the first IOFETCH (If there is to be more than one).
        T ← uiBUFPTR ← 377C;
        uiNBUFPTR ← (uiNBUFPTR)-(357C); *From here on, uiNBUFPTR is used
*for the count (-(NWRDS + (ADDRESS and 17B)))
        T←RHMASK[uiNWRDS];
        uiBUFPTR ← (uiBUFPTR)-(T); *377- number of words for the display
        uiBUFPTR ← (uiBUFPTR) OR (100000C); *Wakeup disable bit
        T← (RHMASK[uiNWRDS])+(T); * T ← 2*NWRDS
        uiDBA←(uiDBA)+(T); *uiDBA is now set up for the next scan line
        uiDWA←T←(uiDWA)+(20C),CALL[uiDWT1];

*Loop for second through Nth IOFETCH.
uiDWT1: uiNBUFPTR←(uiNBUFPTR)+(20C),GOTO[uiBUFD2,R>=0];
        IOFETCH16[uiBASE,uiDBADDR],GOTO[uiBUFD2X,ALU>=0];
        uiDWA←T←(uiDWA)+(20C),RETURN;

uiBUFD2: uiVSCOUNT ← (uiVSCOUNT)-1,DBLGOTO[uiENDFIELD,uiCONT,R<0]; *check for field done
uiBUFD2X: uiVSCOUNT ← (uiVSCOUNT)-1,DBLGOTO[uiENDFIELD,uiCONT,R<0];
uiCONT: uiSLC←(uiSLC)-1,DBLGOTO[uiDCBDONE,uiMDCB2,R<0];


*Calculate the next  line's uiNBUFPTR
*in the shadow of the last IOFETCH16
uiMDCB2: uiNBUFPTR ← 377C;
        T←LDF[uiDBA,14,4];
        uiNBUFPTR ← (uiNBUFPTR)-(T);
        T←RHMASK[uiNWRDS];
        uiNBUFPTR ← (uiNBUFPTR)-(T);
        OUTPUT[uiBUFPTR,uiBPREG], goto[xpreVS0];

*The DCB is finished.
uiDCBDONE:      DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
        T←(uiLINK);
        uiBASE1 ← 0C, GOTO[uiGetNextDCB,ALU#0];
*The DCB chain is exhausted.
        OUTPUT[uiBUFPTR,uiBPREG]; *Send read BUFPTR to the hardware
        T ← 2C, call[uiCheckCursor]; *does TASK return
        goto[uiVS1];


uiGetNextDCB:
        uiBASE ← T; *T contains LINK. Set base register to point to next DCB
        OUTPUT[uiBUFPTR,uiBPREG]; *Send read BUFPTR to the hardware
        uiNBUFPTR ← 377C; *Init for later
        nop;*two instr after output
        PFETCH2[uiBASE,uiDBA,2]; *Fetch DBA,SLC
        uiBUFPTR ← 377C; *Init for later
*Check for long pointer addressing
        PFETCH2[uiBASE,uiLINK,0]; *Fetch Link,NWRDS
        LU←uiSLC,goto[uiLong,R<0];

*Short Pointer
        T ← uiDBA;
        uiBASE ← T,goto[uiEvenOdd];


*Long Pointer
uiLong:
        PFETCH2[uiBASE,uiBASE,4]; *fetch directly into the base register
        uiSLC ← (uiSLC) AND NOT (100000C);  *clear the sign bit


*Bias uiDCB.SLC by -2.  Note that if uiDCB.SLC = 0 OR 1, at least one
*scan line will be displayed.

uiEvenOdd: LU ← LDF[uiCRWORD,17,1];     *Check EvenField
        uiSLC ← (uiSLC)-(2C),GOTO[uiDBAOK,ALU#0];

uiDBABAD: T← RHMASK[uiNWRDS];
        uiBASE ← (uiBASE)+(T);
uiDBAOK: T←LDF[uiBASE,14,4]; *Set up NBUFPTR for the next scan
        uiNBUFPTR ← (uiNBUFPTR)-(T); *uiNBUFPTR ← 377C earlier
        T←RHMASK[uiNWRDS];
        uiNBUFPTR←(uiNBUFPTR)-(T);

*Now fix up the base register so that it is hex aligned and DBA contains the residue

        T ← (uiBASE) and (17C);
        uiDBA ← T;
        uiBASE ← (uiBASE) and not (17C);
*fix up the high half of the base register
        T ← lsh[uiBASE1,10];
        uiBASE1 ← (RHMASK[uiBASE1])+(T)+1;

        T← 2C, call[uiCheckCursor];  *returns to VS2

*We have just picked up a new DCB.  We must output HTAB,
*then go to normal state 0 processing.
```

```
uiVS2:  T ← LDF[uiNWRDS,2,6]; *calculate HTAB
        LU←LDF[uiNWRDS,1,1]; *black background bit
        uiBUFPTR ← (uiBUFPTR) - (T)-1,GOTO[.+2,ALU#0]; *uiBUFPTR ← 377C earlier
        uiBUFPTR ← (uiBUFPTR) AND NOT (200C);
        OUTPUT[uiBUFPTR,uiHTAB],goto[uiVS0]; *send it

uiENDFIELD:     uiBASE1 ← 0C;
        uiCRWORD ← (uiCRWORD) xor (3C), goto[uiFD1];
```

```
*We have displayed the entire chain and the field is not done.
*Wait for end of field.
xpreVS1:        T← 2C, call[uiCheckCursor];
preVS1: call[uiWAKEOFF];
uiVS1: DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
        uiVSCOUNT ← (uiVSCOUNT)-1, GOTO[uiFIELDDONE,R<0];
        goto[xpreVS1];


*Turn off wakeallow and return
uiWAKEOFF:  uiCRWORD ← (uiCRWORD) AND NOT (200C); *AllowWake ← 0
        OUTPUT[uiCRWORD,uiCREG];
        uiCRWORD ← (uiCRWORD) OR (200C);
        uiBUFPTR ← 377C; *set up for next run.

*               OUTPUT[uiCRWORD,uiCREG],return;
        OUTPUT[uiCRWORD,uiCREG];
Outputwait:
        nop;
        return;

*Check for cursor visible.  Enter with T=2.
uiCheckCursor:
        uiCY←(uiCY)-(T),GOTO[uiCURRET,R>=0];
        uiCX ← (uiCX)+(1000C);
        LU ← LDF[uiCX,3,1];
        GOTO[uiSENDCX,ALU=0];
        uiCX ← 5C;         *finished displaying the cursor
        uiCY ← 10000C;
uiSENDCX:
        OUTPUT[uiCX,uiCXREG],goto[Outputwait];
uiCURRET:       return;

*The field is finished.  Make Vsync pulse.
uiFIELDDONE:
        uiCRWORD ← (uiCRWORD) XOR (3C);  *complement field, set PreVS.
uiFD1:  uiBUFPTR ← 200C;
 *send black background so that hsync won't be screwed up
        OUTPUT[uiBUFPTR,uiHTAB], call[uiWAKEOFF]; *returns to uiVS3
```

```
*We are in the first scan line of a vertical sync pulse.
*Post the mouse COORDINATES to core.
uiVS3: uiBASE ← 0C;
        T← (uiBUFPTR)+(25C); *424b AND 425b = Mouse x and y.
        PFETCH2[uiBASE,uiDBA]; *uiDBA and uiSLC are used as temps.
        DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
        uiMOUSEDELXY ← LCY[uiMOUSEDELXY,11],DBLGOTO[uiSEM1,uiNSEM1,R<0];

uiSEM1: T ← 177C,GOTO[uiSEM1FIN];
uiNSEM1: T ← (ZERO)-1;
uiSEM1FIN: T← (LDF[uiMOUSEDELXY,7,7]) XNOR (T);
        uiDBA ← (uiDBA)+(T);

        T←uiMOUSEDELXY←LDF[uiMOUSEDELXY,0,7],DBLGOTO[uiSEM2,uiNSEM2,R<0];

uiSEM2: T ← (uiMOUSEDELXY) XNOR (177C);
uiNSEM2: uiSLC ← (uiSLC)+(T);
        T← (uiBUFPTR)+(25C);
        PSTORE2[uiBASE,uiDBA]; *Restore coordinates
        uiMOUSEDELXY ← (ZERO), call[uiWAKEOFF]; *returns to uiVS4
```

```
*Post the mouse BUTTONS
uiVS4: uiBASE←177000C; *NOTE modification of uiBASE
        uiBASE←(uiBASE)+(30C); *Fetch 177030
        PFETCH1[uiBASE,uiQTEMP,0]; *uiQTEMP overlays LINK, NWRDS, DBA, SLC.
        DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];

*convert UTFP mouse button order into ALTO order
*On the UTFP, the sequence for the buttons (BUTTONS[13:15]) is right,
*middle,left, and 1's mean buttons depressed.
*On the ALTO, 177030[15:17] correspond to left,right,middle, and 1's
*in memory mean button NOT depressed.

        t← ldf[uiBUTTONS,13,1]; *right button
        uiTEMP ← t;
        t ← ldf[uiBUTTONS,14,1]; *middle button
        uiTEMP ← (lsh[uiTEMP,1]) or (t);
        t ← (uiBUTTONS) and (4C); *left button
        t ← (uiTEMP) or (t);
        uiQTEMP ← (uiQTEMP) or (7C);
        t ← uiQTEMP ← (uiQTEMP) xor (t);
        * ignore uiQTEMP1 and uiQTEMP2
        uiQTEMP3 ← t;
        noop;     * wait for write of register
        PSTORE4[uiBASE,uiQTEMP,0];
        uiBASE ← ZERO; *reset base register


*Fetch new dcb chain header and interrupt mask.
        T ← (uiBUFPTR)+(21C); *T←420, Since uiBUFPTR=377.
        PFETCH2[uiBASE,uiLINK]; *Get new dcb header from 420, intmask from 421.


*check for realtime clock update
*save STKP
*       uiTEMP ← 325C; *Point to RTCLOW
        uiTEMP ← 355C; *Point to RTCLOW
        T← nSTKP;
        STKP ← uiTEMP, uiTEMP ← T,NoRegILockOK;
        uiTEMP ← (uiTEMP) XOR (377C); *STKP read inverted
        STACK ←(STACK) AND NOT (100000C),GOTO[uiNORTCOV,R>=0];
*must update RTC
        uiTEMP1←400C;
        T←(uiTEMP1)+(30C);
        PFETCH1[uiBASE,uiTEMP1];
        uiTEMP1 ← (uiTEMP1) + 1;
        PSTORE1[uiBASE,uiTEMP1];
*cause vertical field interrupt
uiNORTCOV:      STKP ← uiTEMP;
        loadpage[0];
        T ← uiNWRDS, callp[DoInt]; *interrupt mask fetched from 421 above

uiNOTMR:        call[uiWAKEOFF]; *Returns to uiVS5
```

```
*Post the keyboard

uiVS5:  lu ← LHMASK[uiXMSG], call[uiKPOST];
        lu ← LHMASK[uiXMSG], call[uiKPOST]; *do it again for other byte
        DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
        uiBASE ← ZERO, GOTO[preVS6];

uiKPOST:
        uiTEMP ← KeyTableH,goto[.+2,alu#0]; *if no data, return right away
        uiXMSG ← lsh[uiXMSG,10],return; *shift to other keyboard char

        uiTEMP ← (uiTEMP) or (KeyTableL);
        t ← ldf[uiXMSG,1,5]; *Get word number (4 bytes per word)
        uiTEMP ← (uiTEMP) + (T); *form final address
        t ← ldf[uiXMSG,6,1]; *set h2 to high/low word
        APC&APCTASK ← uiTEMP; *Address to read in Control Store
        READCS; *get the word
        t ← CSDATA, AT[uiUTFPBASEADDR,300]; *must be at an even location for READCS
        uiDBA ← t;
        lu ← LDF[uiXMSG,7,1]; *low or high byte
        goto[.+2,alu#0], uiBASE ← 177000C;
        uiDBA ← RSH[uiDBA,10]; *Need upper byte
        uiBASE ← (uiBASE)+(34C); *uiBASE ← 177034C
        T←(LDF[uiDBA,15,3]); *Get word number
        PFETCH1[uiBASE,uiNWRDS]; *uiNWRDS is a temp - fetch Alto kbd word
        uiDBA ← LDF[uiDBA,11,4]; *Get bit number
        uiBASE ← (uiBASE)+(T); *fix base register for store
        uiTEMP ← T ← 100000C; *Do the function uiTEMP ← 100000 rshift uiDBA

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 15
        uiTEMP ← T ← RSH[uiTEMP,1]; *shift 1

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 14
        uiTEMP ← T ← RSH[uiTEMP,2]; *shift 2

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 13
        uiTEMP ← T ← RSH[uiTEMP,4]; *shift 4

        uiDBA ← RSH[uiDBA,1], goto[.+2,REVEN]; *test bit 12
        uiTEMP ← T ← RSH[uiTEMP,10]; *shift 8


uiKDD:  *test for key down (0) or up (1)
        uiXMSG ← lsh[uiXMSG,10], DBLGOTO[uiKDOWN,uiKUP,r>=0];

uiKDOWN: uiNWRDS ← (uiNWRDS) AND NOT (T), GOTO[uiKSTORE]; *key down, clear bit
uiKUP: uiNWRDS ← (uiNWRDS) OR (T); *key up, set bit
uiKSTORE: goto[uiMSGRTN1],PSTORE1[uiBASE,uiNWRDS,0]; *store word

preVS6: uiBUFPTR ← 377C, call[uiWAKEOFF]; *returns to uiVS6
```

```
*Set up the cursor.
*Get cursor coordinates from 426b (X), and 427b (Y).
uiVS6: T  ← (uiBUFPTR)+(27C); *uiBUFPTR = 377 on entry.
        PFETCH2[uiBASE,uiCX]; *Cursor X,Y coordinates
        DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];


*Experiment has determined that the cursor is 3 nibbles to the left
*and one scan line below its proper position.  We fudge...
        uiCY ← (uiCY)+1;
        uiCX ← (uiCX)+(14C);

*The even scan lines of the cursor will be displayed if CY is even
*and the field is even, or if CY is odd and the field is odd.
*Otherwise, the odd scan lines will be displayed.
        uiBUFPTR ← (uiBUFPTR)+(32C); *uiBUFPTR ← 431b

*The cursor X counter is loaded by HSF from CXREG.  It is
*clocked by [(EdgeClock and SelCursM) or (NClk and (HSF or VS'))].

*When the cursor is visible, -X is loaded into the CXREG in the
*scan line preceding the cursor, and this value is loaded into
*the cursor counter by HS.  When VS=0, the cursor counter is
*incremented by NClk, and when it becomes 0, the next 5 nibbles are
*sent to the display.

*Here, we want the cursor counter to address the cursor memory so
*that we can load it.  We send 0 to CXREG, and during each of the
*next 8 scan line times we will send 5 bytes of cursor data to the
*memory, then send a new segment value to uiCXREG.

*uiVSCOUNT is used to hold the value to be loaded into uiCXREG.
        uiVSCOUNT ← 1000C; *Start load at segment 1.

        T←LDF[uiCRWORD,17,1];  *Now finish setting up the pointer to the
*cursor area in main storage.
        T← (LDF[uiCY,17,1]) XOR (T);
        T← (uiBUFPTR)+T;
        uiDBA ← T, goto[uiMORECSETUP];
```

```
uiMORECSETUP: OUTPUT[uiVSCOUNT,uiCXREG], call[uiWAKEOFF];

*Load one segment of the cursor memory (5 bytes plus one zero byte).
*The address was set up during the previous scan line.
uiVS7: T ← uiDBA;  *Pointer to cursor segment
        PFETCH1[uiBASE,uiTEMP]; *Fetch segment
        DISPATCH[uiMPSTATUS,5,3],CALL[uiCHKMSG];
        uiDBA ← (uiDBA)+(2C); *Increment pointer (by 2 due to interlace)
        DISPATCH[uiCX,16,2];     *Determine amount to shift word.
        DISP[uiSHC0],uiTEMP1 ← 17C;

*Subroutine for loading the cursor memory:
uiSENDCUR: uiNWRDS ← T, usectask; *uiNWRDS is a temporary, not used during VS.
        OUTPUT[uiNWRDS,uiCURSM], return;

uiSHC0: T← 0C, GOTO[uiSHCDONE], AT[uiSHCUR,0];
uiSHC1: T ← LSH[uiTEMP,3], AT[uiSHCUR,1];
        uiTEMP ← RSH[uiTEMP,1], GOTO[uiSHCDONE];
uiSHC2: T←LSH[uiTEMP,2], AT[uiSHCUR,2];
        uiTEMP ← RSH[uiTEMP,2],GOTO[uiSHCDONE];
uiSHC3: T ← LSH[uiTEMP,1], AT[uiSHCUR,3];
        uiTEMP ← RSH[uiTEMP,3], GOTO[uiSHCDONE];

uiSHCDONE: uiTEMP1 ← (uiTEMP1) AND (T);
        uiCNT ← 2C;
uiSENDCLOOP: T← LDF[uiTEMP,0,4], CALL[uiSENDCUR];  *Loop for first 4 bytes
        uiCNT ← (uiCNT)-1, GOTO[uiDONECUR,R<0];
        uiTEMP ← LSH[uiTEMP,4],GOTO[uiSENDCLOOP];

uiDONECUR:  OUTPUT[uiTEMP1,uiCURSM]; *Send 5th byte.
        LU← LDF[uiVSCOUNT,3,1];
        OUTPUT[uiTEMP,uiCURSM],GOTO[uiCSETUPDONE,ALU#0]; *TEMP is 0 (6th byte)
        uiVSCOUNT ← (uiVSCOUNT)+(1000C),GOTO[uiMORECSETUP];    *Increment segment address

uiCSETUPDONE:  T← LDF[uiCX,6,10]; *CX counts nibbles, not bits
        uiCX ← (ZERO)-T; *And it is negated.
        LU←uiCRWORD, goto[uiCSDONE,RODD]; *Test field
        uiCY←(uiCY)-(1C);
uiCSDONE:       uiVSCOUNT←5C;
        OUTPUT[uiVSCOUNT,uiCXREG], call[uiWAKEOFF]; *returns to uiVS10
```

```
*We are in the last scan line of a vertical sync pulse.
*Set up uiVSCOUNT for the next field.
*uiLINK has rv420, fetched during VS4.

uiVS10: uiCRWORD ← (uiCRWORD) AND NOT (2C); *PreVS ← 0
        OUTPUT[uiCRWORD,uiCREG];
        uiBUFPTR ← (uiBUFPTR) OR (100000C);
        LU ← LDF[uiCRWORD,17,1];
        T ← uiLINESPERFIELD,DBLGOTO[uiEVX,uiODX,ALU=0];
uiEVX: uiVSCOUNT ← T, GOTO[uiDCBDONE];
uiODX: uiVSCOUNT ← (ZERO)+(T)+1, GOTO[uiDCBDONE];

        end[uitask];
```

```
Title[XWDefs];  *  Definitions for Xerox wire microcode

* Last modified by Murray on September 15, 1979  12:03 PM
*    Base reg changes
*  modified by Murray on September 13, 1979  7:47 PM
*  modified by Chang on August 22, 1979  6:00 PM, move Timer's regs & CSB
*  modified by Chang on August 10, 1979  8:18 AM, Re-Change CSB
*  modified by Chang on August 3, 1979  12:00 PM, Change CSB Assignments
*  modified by Chang on June 25, 1979  9:36 PM, EONotify2 = 344
*  modified by Roy Ogus on June 13, 1979  12:25 PM

        SET TASK [0]; *For R addressing

%
* Defs in GlobalDefs
Set[EITask, *];  * Wire input task number
Set[EOTask, *];  * Wire output task number
Set[EITask2, *];  * Wire input task number (second controller)
Set[EOTask2, *];  * Wire output task number (second controller)
Set[EEPage, *];  * Emulator (for SIO) page number
Set[EIPage, *];  * Wire input page number
Set[EOPage, *];  * Wire output page number
%

*Dispatch table locations
Set[EEBase, LSHIFT[EEPage, 10]];
Set[EESIOLoc, ADD [EEBase, 120]];        * Dispatch location for SIO (bits 16, 17)
Set[EE2SIOLoc, ADD [EEBase, 140]];       * Dispatch location for SIO (bits 14, 15)
* Note following def in GlobalDefs:
*Set[RS232SIOLoc, add[LShift[EEPage,10], 370]];

*Address constants
Set[EIStartLoc, ADD[LSHIFT [EIPage, 10], 160]];          * Input notify location
Set[EOStartLoc, ADD[LSHIFT [EOPage, 10], 130]];          * Output notify location
Set[EIStartLoc2, ADD[LSHIFT [EIPage, 10], 164]];                 * Input notify location
Set[EOStartLoc2, ADD[LSHIFT [EOPage, 10], 134]];                 * Output notify location
Set[EOTimerDoneLoc, ADD[LSHIFT [EOPage, 10], 110]];      * Output TimerDone notify location


* Use to write state from Task 0
MC[XOWriteState,OR@[LSHIFT[EOTask,4],0]];
MC[XIWriteState,OR@[LSHIFT[EITask,4],0]];
MC[XOWriteState2,OR@[LSHIFT[EOTask2,4],0]];
MC[XIWriteState2,OR@[LSHIFT[EITask2,4],0]];

* Constants to define microcode type.  Mask used for SIO return.
MC[AltoEtherMask, 77400];        * Alto Ethernet emulation
MC[EtherIOCBMask, 37400];        * Ethernet hardware IOCB microcode
MC[XWireIOCBMask, 57400];        *  Xerox Wire hardware IOCB microcode


* I/O Address Registers
Set[XIData, 3];                  * Input data
Set[EIHost, 1];                  * Host ID (Ethernet only)
Set[XWStatus, 2];                * Status/State register
Set[XOData, 1];                  * Output data
Set[XWReadState, 2];             * State register read
Set[XWWriteState, 0];            * State register write

*  REGISTER definitions
*  See XWTask.mc for description of register usage.
*  These are task specific registers,
*     there is an identical set for input and output task.
*  Note:  reg. 0 and 1 used by DoInt.
RV[InitialCSB, 0]; * CSB pointer passed in this reg from initialize.mc
RV[XWTemp, 2];
RV[XWCompletion, 3];  * Not used very often
RV[XWIndex, 4];  * Quad word fetch
RV[XWCount, 5];  * These must match the layout in the IOCB
RV[XWPtr, 6];   RV[XWPtrHi, 7];
RV[XWIOCB, 10];
RV[XWIOCBHi, 11];
RV[XWCSB, 12];
RV[XWCSBHi, 13];
RV[XWTemp1, 14];  RV[XWTemp2, 15];  RV[XWTemp3, 16];  RV[XWTemp4, 17];
*  Register definitions for first 4 words of CSB (overlay others)
RV[XWCSB1, 4];  RV[XWNextIOCB, 5];  RV[XIHALo, 6];  RV[XIHAHi, 7];

* State Register command words
MC[XWSetPurgeMode, 260];  * Sets: Enable Input, PurgeMode
MC[XWSetOutputEOP, 107];  * Sets: Enable Output, OutputEOP, JamEnable
MC[XWEnableInput, 220];  * Sets: Enable Input
MC[XWEnableOutput, 103];  * Sets: Enable Output, JamEnable
MC[XWDisableInputOutput, 300];  * Clears: Input and Output
MC[XWDisableInput, 200];  * Clears: Input state register
MC[XWDisableOutput, 100];  * Clears: Output state register
MC[XWPream, 252];  * Xerox Wire preamble

*Status bit definitions: as read with Read State/status command.
* Status bits (Ethernet)
SET [XSIJam, 100000];            * Receiver-detected collision (Jam)
SET [XSOURun,  40000];           * Output Underrun
SET [XSIORun,  20000];           * Input overrun
SET [XSOCOLL,  10000];           * Transmitter-detected collision (Collision)
SET [XSICRC,   4000];            * Input Bad CRC
SET [XSOFAULT,  2000];           * Output Data Fault
SET [XSOPAR,  1000];             * Output Bad Parity
SET [XSIBA,   400];                      * Input Bad Alignment

MC[XISMASK, OR@[XSIJam, XSIORun, XSICRC, XSIBA]];       * Status bits reported for input command
```

```
MC[XOSMASK, OR@[XSOURun, XSOCOLL, XSOFAULT, XSOPAR]];          * Output Status bits
MC[XOCollisionMask, XSOCOLL];

* Status bits (Xerox Wire)
SET [XXSIBA, 100000];              * Input Bad Alignment
SET [XXSIORun, 40000];             * Input overrun
SET [XXSIJam, 20000];              * Receiver-detected collision (Jam)
SET [XXSICRC, 10000];              * Input Bad CRC
SET [XXSOPAR, 4000];               * Output Bad Parity
SET [XXSOURun, 2000];              * Output Underrun
SET [XXSOCOLL, 1000];              * Transmitter-detected collision (Collision)
SET [XXSOFAULT, 400];              * Output Data Fault

MC[XXISMask, OR@[XXSIJam, XXSIORun, XXSICRC, XXSIBA]];  * Status bits reported for input command
MC[XXOSMask, OR@[XXSOURun, XXSOCOLL, XXSOFAULT, XXSOPAR]];  * Output Status bits
MC[XXOCollisionMask, XXSOCOLL];


* Completion codes
MC[XWGoodPacket, 040000];
MC[XWErrorBadHWPkt, 070000];
MC[XWErrorZeroBuf, 064000];                * On input (length<3) and output (length=0)
MC[XWPktBufOverrun, 061000];               * Input only
MC[XWErrorCountOV, 062000];                * Output only


* Timer definitions
Set [XWTimerRunning, 5];  * State 5 is simple timer
Set [XWTimerIdle, 4];  * State 4 is idle
MC[XWTimerMask, LSHIFT [XWTimerRunning, 14]];
MC[XWIdleTimer, LSHIFT [XWTimerIdle, 14]];
MC[pXWRandomReg, 377];  * Number of random number (REFR register)
MC[pEONotifyReg, 340];  * Register used for timer notify
MC[pEONotifyReg2, 344];  * Register used for timer notify (second board)


* Indices into blocks

*  IOCB's
MC[XWIndexNext, 0]; * (Short) Pointer to next IOCB
MC[XWIndexTransInt, 1]; * Retransmission Mask
MC[XWIndexCompletion, 3];
MC[XWIndexBuffer, 4];  * Long pointer and length (Quad word aligned)

*  CSB's
MC[XWIndexHeader, 0]; * First 4 words of CSB
MC[XWIndexPtr, 1];  * Pointer to current IOCB
MC[XWIndexWake, 4];  * Wakup Bitmask
MC[XWIndexNoICBCount, 5];  * Count of times when no ICB chain (For test only)
MC[XWIndexScratch, 10];  * Quad word scratch area

* MACROs: The attn on the XWire is backwards to avoid the BranchBurp
M@[$SkipNOATTEN, IFE@[XWire, 0, SKIP[NOATTEN], SKIP[IOATTEN]]];
M@[$GoToIOATTEN, IFE@[XWire, 0, GOTO[#1, IOATTEN], GOTO[#1, NOATTEN]]];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[XWDefs];

        title[XWInit];

*Last modified by Murray on September 15, 1979  11:22 AM
*  Merge in XWInit2 and Base Register changes
* modified by Murray on September 14, 1979  12:48 AM
* modified by Roy Ogus on June 13, 1979  12:27 PM

 *For register addressing.  This code actually runs at the assigned task level.
        SET TASK [0];

*  ETHERNET controller INITIALIZATION.
*  Will only be called if there is an Ethernet board in the machine.
*  Read Ethernet ID from board and form constant to be returned by SIO.
*  Setup Notify value in EONotifyReg

        ON PAGE [EtherInitPage];

XIInit: CALL [XWSetup], XWTemp ← pEHostRegx, AT[EtherInInitLoc];

        Input[Stack,EHost];
        Stack ← (Stack) AND (377C);      * HostNumber in right half
        Stack ← (Stack) OR (LFE@[XWire, 0, EtherIOCBMask, XWireIOCBMask]);
        RETURN, Stkp ← XWTemp4; *restore Stkp


XOInit: CALL [XWSetup], XWTemp ← pEONotifyReg, AT[EtherOutInitLoc];

*  Compute value for EONotify register, used for notify after timer wakeup.
        XWTemp ← AND@[0377, EOTimerDoneLoc]C;    * Low 8 bits of APC
        XWTemp ← (XWTemp) OR (OR@[lshift[EOTask,14],AND@[7400,EOTimerDoneLoc]]C); * High 4 bits of APC

        T ← XWTemp;
        Stack ← T;
        RETURN, Stkp ← XWTemp4; *restore Stkp


* Second Board

XIInit2:        CALL [XWSetup], XWTemp ← pEHostRegx, AT[EtherInInitLoc2];
        RETURN, Stkp ← XWTemp4; *restore Stkp


XOInit2:        CALL [XWSetup], XWTemp ← pEONotifyReg2, AT[EtherOutInitLoc2];

*  Compute value for EONotify register, used for notify after timer wakeup.
        XWTemp ← AND@[0377, EOTimerDoneLoc]C;    * Low 8 bits of APC
        XWTemp ← (XWTemp) OR (OR@[lshift[EOTask2,14],AND@[7400,EOTimerDoneLoc]]C); * High 4 bits of APC
        T ← XWTemp;
        Stack ← T;
        RETURN, Stkp ← XWTemp4; *restore Stkp


XWSetup:
        T ← InitialCSB;
        XWCSB ← T;
        XWCSBHi ← 0C;
        XWIOCBHi ← 0C;

        T ← GETRSPEC[103] xor (377C); *Stkp (inverted)
        UseCTask, XWTemp4 ← T;
        RETURN, Stkp ← XWTemp;

        end[xwireinit];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[XWDefs];

        title[XWInit2];

*Last modified by Roy Ogus on June 22, 1979  3:55 PM

        SET TASK [0]; *For register addressing.  This code really runs at task level EITask

*  ETHERNET controller INITIALIZATION.
*  This subroutine is for the second controller board (executed by EITask).
*  Will only be called if there is a second Ethernet board in the machine.
*  Will only be called after init of first controller (which set host address).
*  Initializes the notify register.

        ON PAGE [EtherInitPage];

XWireInit2:     T ← GETRSPEC[103] xor (377C), at[EtherInitLoc2]; *Stkp (inverted)
*  Compute value for EONotify2 register, used for notify after timer wakeup.
        XWBase ← AND@[0377, EOTimerDoneLoc]C;    * Low 8 bits of APC
        XWBase ← (XWBase) OR (OR@[lshift[EOTask2,14],AND@[7400,EOTimerDoneLoc]]C); * High 4 bits of APC

        XWBaseHi ← pEONotifyReg2;
        Stkp ← XWBaseHi, XWBaseHi ← T;
        T ← XWBase;
        Stack ← T;
        RETURN, Stkp ← XWBaseHi; *restore Stkp
*Note - base registers are initialized each time input or output is disabled, so we
*do not need to do it here as long as input and output are disabled before enabled.

        end[xwireinit2];
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[XWDefs];
TITLE [XWSIO];  * Xerox Wire microcode (Defs in XWDefs)
                * Written by Roy Ogus.
%
        Last modified by Roy Ogus - June 13, 1979  12:31 PM

Notes:  - This version handles 1 Ethernet or 1 Xerox wire board.
        - This module has SIO code for XWire.
        - Contains Conditional assembly.
            XWire=0 for Ethernets, XWire=1 for Xerox Wire boards.

Log:    - File reorganization (January 15, 1979  11:23 AM).
        - Conditional assembly (February 4, 1979  5:30 PM).
        - Upgrade to 3.0 (April 12, 1979  5:14 PM).
%
        SET TASK [0];

*
*
* EMULATOR TASK -- SIO instruction

        ON PAGE [EEPage];

*This code executes at task 0
* We get here after the SIO instruction has been issued.
*  The SIO control bits are in AC0.
* Two-board format (using IOCBs):
*  SIO bits 16, 17:  Board 1.
*       0  -  NOP
*       1  -  Disable output (TPC is initialized).
*       2  -  Disable input (TPC is initialized).
*       3  -  Disable board 1 (TPC not initialized)
*  SIO bits 14, 15:  Board 2.
*       0  -  Unused (NOP)
*       1  -  Disable output (TPC is initialized)
*       2  -  Disable input (TPC is initialized)
*       3  -  Disable board 2 (TPC not initialized)
*
*  Note: RS232 SIO bits are 10, 11.

* To enable boards, use OUTPUT[Command, StateRegister];
* Board 1 : StateRegister1 = LSHIFT[EOTask, 4].
* Board 2 : StateRegister2 = LSHIFT[EOTask2, 4].
* Commands:
*       EnableOutput = 103B.
*       EnableInput = 220B.
*
* SIO returns (in AC0):
*       AC0[0:2]  = 3  -  Alto-emulation Ethernet microcode
*                 = 2  -  Xerox Wire hardware, IOCB microcode
*                 = 1  -  Ethernet hardware, IOCB microcode
*       AC0[3:7] = 37B
*       AC0[10B:17B]  =  Host number (Ethernet)
*
* (Interim) Controller Status Blocks:
*   Board 1:  OCSB  at 177200B,  ICSB at 177220B.
*   Board 2:  OCSB  at 177240B,  ICSB at 177260B.

* This code is really part of the emulator, and uses its temporary registers.
* RTEMP1 = 1 if called from Mesa, 0 if called from Nova.
* Get host address constant for Ethernet.
EESIO:  AC0 ← T, at[EEStartLoc];        *save control bits (useful only if called from Mesa)
* RS232 SIO command.
        T ← LDF[AC0,10,2];      * RS232 SIO bits are 10, 11
        RTemp ← AND@[377, RS232SIOLoc]C;
        RTemp ← (RTemp) OR (OR@[LSHIFT[16,14],AND@[007400, RS232SIOLoc]]C);
        RTemp ← (RTemp) OR (T);
        APC&APCTask ← RTemp, CALL [XETaskRet];
*       Return;
* End RS232 SIO command.
        T ← 37C;                * check bits 3:7
        T ← (ldf[EHostReg,3,5]) xor (T); *these bits will be 37b if the init code was run
*(i.e. if there is an Ethernet board in the machine), and will be zero otherwise
        T ← EHostReg, goto[EESIODisp,ALU=0];
        AC0 ← 0C;
        AC0 ← (AC0) xnor (100000C),goto[EESIODone]; *return 77777b in AC0 if no Ethernet board
*
* First Ethernet board (Dispatch on bits 16,17):
EESIODisp:      DISPATCH [AC0, 16, 2];
        DISP [EESIO0], AC0 ← T;         * This line when only one controller
*
* 00 -- Do nothing.
EESIO0: lu ← RTEMP1, dblgoto[EEMesaRet,EENovaRet,Rodd], AT[EESIOLoc, 0];
* 01 -- Disable output.
* Form APC&APCTask word to notify the output microcode
EESIO1: RTEMP ← AND@[0377, EOStartLoc]C, AT[EESIOLoc, 1];
        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EOTask,14],AND@[007400, EOStartLoc]]C);
* 10 -- Disable Input
* Form APC&APCTask word to notify the input microcode
EESIO2: RTEMP ← AND@[0377, EIStartLoc]C, AT[EESIOLoc, 2];
        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EITask,14],AND@ [007400, EIStartLoc]]C);
* 11 -- Disable Input and Output (TPC not initialized).
EESIO3: GOTO[EESIOAbort], RCNT ← XWDisableInputOutput, AT[EESIOLoc, 3];

* Notify appropriate code (disable done by the task code).
EESIONotify:    CALL[XETaskRet], APC&APCTASK ← RTEMP;

* Return here when emulator runs again.
```

```
EESIODone:      lu ← RTEMP1, dblgoto[EEMesaRet,EENovaRet,Rodd];

EENovaRet:      loadpage[nePage];
        FF1@[17], gotop[neNoskip];

EEMesaRet:      loadpage[7];
        gotop[P7fail];
*
* Used for tasking.
XETaskRet:      RETURN;


*
* Code for STC[3]. Disable input and output on boards.
EESIOAbort:     T ← XOWState;
        GOTO [EESIODone], OUTPUT[RCNT];
        END;
```

```
* modified by HGM - September 6, 1979  7:34 AM
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[XWDefs];
TITLE [XW2SIO]; * Xerox Wire microcode (Defs in XWireDefs)
                * Written by Roy Ogus.
%
        Last modified by Roy Ogus - June 22, 1979  4:15 PM

Notes:  - This version handles 2 Ethernets or 2 Xerox wire boards.
        - This module has SIO code, and extra code for 2nd controller.
        - Contains Conditional assembly.
            XWire=0 for Ethernets, XWire=1 for Xerox Wire boards.

Log:    - File reorganization (January 15, 1979  11:23 AM).
        - Conditional assembly (February 4, 1979  5:30 PM).
        - Upgrade to 3.0 (April 12, 1979  5:14 PM).
%
        SET TASK [0];

*
*
* EMULATOR TASK -- SIO instruction

        ON PAGE [EEPage];

*This code executes at task 0
* We get here after the SIO instruction has been issued.
*  The SIO control bits are in AC0.
* Two-board format (using IOCBs):
*  SIO bits 16, 17:  Board 1.
*       0  -  NOP
*       1  -  Disable output (TPC is initialized).
*       2  -  Disable input (TPC is initialized).
*       3  -  Disable board 1 (TPC not initialized)
*  SIO bits 14, 15:  Board 2.
*       0  -  Unused (NOP)
*       1  -  Disable output (TPC is initialized)
*       2  -  Disable input (TPC is initialized)
*       3  -  Disable board 2 (TPC not initialized)
*
*  Note: RS232 SIO bits are 10, 11.

* To enable boards, use OUTPUT[Command, StateRegister];
* Board 1 : StateRegister1 = LSHIFT[EOTask, 4].
* Board 2 : StateRegister2 = LSHIFT[EOTask2, 4].
* Commands:
*       EnableOutput = 103B.
*       EnableInput = 220B.
*
* SIO returns (in AC0):
*       AC0[0]  = 0  -  Alto-emulation Ethernet microcode
*       AC0[1:2]  = 3  -  Alto-emulation Ethernet microcode
*                 = 2  -  Xerox Wire hardware, IOCB microcode
*                 = 1  -  Ethernet hardware, IOCB microcode
*       AC0[3:7] = 37B
*       AC0[10B:17B]  =  Host number (Ethernet)
*

* This code is really part of the emulator, and uses its temporary registers.
* RTEMP1 = 1 if called from Mesa, 0 if called from Nova.
* Get host address constant for Ethernet.
EESIO:  AC0 ← T, at[EEStartLoc];        *save control bits (useful only if called from Mesa)
* RS232 SIO command.
        T ← LDF[AC0,10,2];       * RS232 SIO bits are 10, 11
        RTemp ← AND@[377, RS232SIOLoc]C;
        RTemp ← (RTemp) OR (OR@[LSHIFT[16,14],AND@[007400, RS232SIOLoc]]C);
        RTemp ← (RTemp) OR (T);
        APC&APCTask ← RTemp, CALL [XWReturn];
* End RS232 SIO command.

* Check bits 3:7,  They will be 37b if the init code was run
* (i.e. if there is an Ethernet board in the machine)
* and will be zero otherwise.
        T ← 37C;
        T ← (ldf[EHostReg,3,5]) xor (T);
        T ← EHostReg, goto[EESIODisp,ALU=0];
        AC0 ← 0C;
        AC0 ← (AC0) xnor (100000C),goto[EESIODone];
* Return 77777b in AC0 if no Ethernet board

* First Ethernet board (Dispatch on bits 16,17):
EESIODisp:
        DISPATCH [AC0, 16, 2];
*       DISP [EESIO0], AC0 ← T; * This line when only one controller
        DISP [EESIO0];          * This line when more than one controller

* 00 -- Do nothing.
EESIO0: GOTO[EE2SIOA], DISPATCH [AC0, 14, 2], AT[EESIOLoc, 0];

* 01 -- Disable output.
* Form APC&APCTask word to notify the output microcode
EESIO1: RTEMP ← AND@[0377, EOStartLoc]C, AT[EESIOLoc, 1];
        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EOTask,14],AND@[007400, EOStartLoc]]C);

* 10 -- Disable Input
* Form APC&APCTask word to notify the input microcode
EESIO2: RTEMP ← AND@[0377, EIStartLoc]C, AT[EESIOLoc, 2];
        GOTO [EESIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EITask,14],AND@ [007400, EIStartLoc]]C);
```

```
* 11 -- Disable Input and Output (TPC not initialized).
EESIO3: GOTO[EESIOAbort], RCNT ← XWDisableInputOutput, AT[EESIOLoc, 3];

* Notify appropriate code (disable done by the task code).
EESIONotify:
        CALL[XWReturn], APC&APCTASK ← RTEMP;

* Second Ethernet board (Dispatch on bits 14,15):
EE2SIO: DISPATCH [AC0, 14, 2];
EE2SIOA:        DISP [EE2SIO0];

* 00 -- Do nothing
EE2SIO0:        GOTO[EESIODone1], T ← EHostReg, AT[EE2SIOLoc, 0];

* 01 -- Disable output.
EE2SIO1:        RTEMP ← AND@[0377, EOStartLoc]C, AT[EE2SIOLoc, 1];
        GOTO [EE2SIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EOTask2,14],AND@[007400, EOStartLoc2]]C);

* 10 -- Disable Input
EE2SIO2:        RTEMP ← AND@[0377, EIStartLoc]C, AT[EE2SIOLoc, 2];
        GOTO [EE2SIONotify], RTEMP ← (RTEMP) OR (OR@[lshift[EITask2,14],AND@ [007400, EIStartLoc2]]C);

* 11 -- Disable Input and Output (TPC not initialized).
EE2SIO3:        GOTO[EE2SIOAbort], RCNT ← XWDisableInputOutput, AT[EE2SIOLoc, 3];

* Notify appropriate code (task code does disable).
EE2SIONotify:
        CALL[XWReturn], APC&APCTASK ← RTEMP;

* Return here when emulator runs again.
EESIODone:      T ← EHostReg;    * Fix AC0 for return
EESIODone1:
        AC0 ← T;
        lu ← RTEMP1, dblgoto[EEMesaRet,EENovaRet,Rodd];

EENovaRet:      loadpage[nePage];
        FF1@[17], gotop[neNoskip];

EEMesaRet:
        loadpage[7];
        gotop[P7Tail];


* Code for SIO[3]. Disable both input and output.
EESIOAbort:
        RTEMP ← AND@[EOTask,17], CALL [XWKillTimer];
        T ← XOWriteState, CALL[XWDisable];
        T ← XIWriteState, CALL[XWDisable];
        GOTO [EE2SIO];

EE2SIOAbort:
        RTEMP ← AND@[EOTask2,17], CALL [XWKillTimer];
        T ← XOWriteState2, CALL[XWDisable];
        T ← XIWriteState2, CALL[XWDisable];
        GOTO [EESIODone];

XWDisable:
        OUTPUT[RCNT], GOTO[XWDally];

XWKillTimer:
        RTEMP ← (RTEMP) + (XWIdleTimer);
        LoadTimer[RTEMP], RETURN;

        END;
```

```
insert[d0lang];
NOMIDASINIT;LANGVERSION;MULTDIB;
insert[GlobalDefs];
insert[XWDefs];
TITLE [XWTask]; * Xerox Wire microcode (Defs in XWDefs)

* Last modified by Murray on September 21, 1979  3:01 AM
*  modified by Murray on September 15, 1979  Base reg changes
*  modified by Murray on September 13, 1979  7:47 PM
*  written by Roy Ogus (early 79).
%

This module contains basic code for a single controller.
File XWSIO2 contains StartIO code for two boards.
This version has Ethernet address recognition.
Data buffers must be quadword-aligned, and at least 4 words long.
Contains conditional assembly code for XWire/Ethernet.
          XWire=0 for Ethernet, XWire=1 for Xerox wire hardware.
Prefetching of CSB at end of packet.
IOATTEN sense is reversed for XWire.

Currently, everything (input, output, and StartIO) fits on a single page.

R register ALLOCATION

R0       ----              (used by DoInt)
R1       ----              (used by DoInt)
R2       XWTemp            - Temporary register
R3       XWCompletion      - Completion word

R4       XWIndex           - Buffer displacement
R5       XWCount           - Buffer word counter (quadword buffer)
R6       XWPtr             - Buffer ptr (low) (current IOCB)
R7       XWPtrHi           - Buffer ptr (high) (current IOCB)

R10      XWIOCB            - Base register for current IOCB (low)
R11      XWIOCBHi          - Base register for current IOCB (high)
R12      XWCSB             - Base register for CSB (low)
R13      XWCSBHi           - Base register for CSB (high)

R14      XWTemp1           - Temporary register (quadword buffer)
R15      XWTemp2           - Temporary register
R16      XWTemp3           - Temporary register
R17      XWTemp4           - Temporary register

R registers that overlay normal use
R4       XWCSB1            - first word of CSB (not currently used)
R5       XWNextIOCB        - Short pointer to next IOCB
R6       XIHALo            - Host address (low - input only)
R7       XIHAHi            - Host address (high - input only)
```

```
        INPUT  MICROCODE

  Input CSB

00      (reserved for emulator control)
01      Short pointer to First IOCB
02      Host address (low word)
03      Host address (high word)
04      Input wakeup bit mask
05      Packets missed (for debugging)
06-07   (unused)
10-13   quadword scratch bufer
14-17   Used by test program

Input IOCB

00      Link to next IOCB
01      (unused)
02      Command word
03      Completion word
04      Amount used in buffer
05      Length of buffer
06      Low pointer to buffer
07      High pointer to buffer
%

        SET TASK [0];    * For R-register addressing

        ON PAGE[EIPage];

* Input microcode is notified at XIStart to initialize.
XIStart:        XWTemp ← XWDisableInput, CALL[XWOutState], AT [EIStartLoc];

*  ADDRESS RECOGNITION: Ethernet encapsulation assumed.
*    Check if the packet,
*      is explicitly for this host (dest=us),
*      or is broadcast (dest=0),
*      or if the host is promiscuous (us=0).

* There are two wakeup points.
*  XIBegin: No prefetch of CSB has been done.
*  XIFastBegin: Prefetch of CSB has been done.
XIBegin:        PFetch4 [XWCSB, XWCSB1, XWIndexHeader!];  * Fetch first 4 words of CSB
XIFastBegin:
        IOStore4 [XWCSB, XIData, XWIndexScratch!];  * Read in 4 words to scratch area in CSB
        LU ← RHMASK[XIHALo];
        SKIP [ALU#0], PFetch4 [XWCSB, XWTemp1, XWIndexScratch!];
        GOTO [XIForMe], T ← XWNextIOCB;  * we are promiscuous
        T ← RSH [XWTemp1, 10];  * Right justify destination host in T
        SKIP [ALU#0], LU ← (RHMASK[XIHALo]) XOR (T);
        GOTO [XIForMe], T ← XWNextIOCB;  * Broadcast
        SKIP [ALU#0];
        GOTO [XIForMe], T ← XWNextIOCB;  * Packet explicitly for me

* Packet not accepted by filter.
*  Tell the hardware to ignore the rest of the packet, i.e. purge packet.
XIPurge:        XWTemp ← XWSetPurgeMode; CALL[XWOutState];
* Return here after wakeup
        GOTO [XIFastBegin];


*  GET BUFFER STARTED
* Check for no IOCB available (in case it was a slow wakeup).
XIForMe:
        GOTO[XIIndexOK, ALU#0], XWIOCB ← T;  * ALU=0 => No IOCB
*** No IOCB.  Bump counter in CSB.
        PFetch1[XWCSB, XWTemp, XWIndexNoICBCount!];
        XWTemp ← (XWTemp)+1;
        PStore1[XWCSB, XWTemp, XWIndexNoICBCount!];
XIPurgeSlow:
        XWTemp ← XWSetPurgeMode; CALL[XWOutState];
* Return here after wakeup
        GOTO [XIBegin];

* Set up IOCB.
XIIndexOK:
* Check for runt packets (<= 3 words, excluding CRC).
* IOATTEN=1 (0 on XWIre) => EOP => small packet
        $SkipNOATTEN;
        GOTO [XIPurge];
        NOP; * Alignment(?) Ether
        CALL[XWBSetup];
        XWCount ← (XWCount)-(4C);
        SKIP [ALU>=0];
        GOTO [XIMark], T ← XWCompletion ← XWErrorZeroBuf;
        NOP; * Alignment(?) XWire
        T ← 0C, CALL[XWStoreTemps];
        CALL[XILoop], XWCount ← (XWCount)-(4C);
*  Main loop-checks for end of buffer, then checks ATTEN for end-of-packet.
XILoop: GOTO [XIQuadFull, R<0], XWCount ← (XWCount)-(4C);
        $SkipNOATTEN, T ← (XWIndex) ← (XWIndex)+(4C);
        GOTO [XIAttn], INPUT[XWTemp, XWStatus];
        RETURN, IOStore4[XWPtr, XIData];

*  Get here when there is no more room for quadwords in the buffer.
*  Check ATTEN to see if end of packet.
XIQuadFull:

* SPIN to scope the weird glitch
*       CALL[.+1];
```

```
*       (XWIndex) ← (XWIndex)+(4C);
*       $GoToIOATTEN [.+2];
*       RETURN, IOStore4 [XWCSB, XIData, XWIndexScratch];
*       GOTO [XIAttn], INPUT[XWTemp, XWStatus];

* Number of singles remaining in buffer = XWCount+8.
* Set up XWCount as (No. singles-1), and read in singles.
        XWCount ← (XWCount)+(7C);
        CALL[XISingles], (XWIndex) ← (XWIndex)+(3C);
XISingles:      GOTO [XIBufFull, R<0], XWCount ← (XWCount)-1;
        $SkipNOATTEN, T ← (XWIndex) ← (XWIndex)+1;
        GOTO [XIAttn], INPUT[XWTemp, XWStatus];
        INPUT [XWTemp, XIData];
        LU ← XWTemp;
        RETURN, PStore1 [XWPtr, XWTemp];

* We get here when the input buffer is exactly full.
* Check if ATTEN for EOP, indicating that the last word was the CRC.
XIBufFull:      $SkipNOATTEN, XWIndex ← (XWIndex)+1;
        GOTO [XIAttn], INPUT[XWTemp, XWStatus];

*  Check if there is only one more word (i.e. the CRC),
*       otherwise mark the packet buffer overrun.
        INPUT [XWTemp, XIData], CALL[XWReturn];
        NOP;  * Wait for ATTN
        $SkipNOATTEN, XWIndex ← (XWIndex)+1;
        GOTO [XIAttn], INPUT [XWTemp, XWStatus];  * EOP, thus there was only one more word (CRC)
        GOTO [XIMark], T ← XWCompletion ← XWPktBufOverrun;


*  FINISH UP  PACKET

*  We arrive here after an IOATTEN/NOATTEN is detected,
*   indicating an end of packet.
*  XWIndex has the number of words stored in the current buffer.
*  XWTemp has status word
XIAttn: T ← LDF [XWTemp, 10, 2];  * Get excess count
        XWIndex ← (XWIndex)-(T)-1, CALL[XWStoreIndex];  * CRC too
XIAttn1:        XWTemp ← (XWTemp) AND (IFE@[XWire, 0, XISMask, XXISMask]);  * Isolate InStatus
        Skip[ALU#0];
        GOTO[XIMark], T ← XWCompletion ← XWGoodPacket;
        XWCompletion ← XWErrorBadHWPkt;  * The hardware indicated problems
        T ← (RSH[XWTemp, 10]);  * Right justify status (Ethernet)
XIMark: XWCompletion ← (XWCompletion) OR (T), CALL[XWStoreStatus];
        CALL[XWNotify];

        CALL[XWNext];
* Now check if there is a buffer.
        T ← XWNextIOCB;
        SKIP[ALU#0], XWIOCB ← T;        * ALU=0 => no IOCB
        GOTO [XIPurgeSlow];
        GOTO [XIPurge];
```

```
% --------------------- OUTPUT  MICROCODE ------------------------------

  Output CSB

00      (reserved for emulator control)
01      Short pointer to First IOCB
02-03   (unused)
04      Output wakeup bit mask
5-17    (unused)

Output IOCB

00      Link to next IOCB
01      Initial Retransmission Interval
02      Command word
03      Completion word
04      (Unused)
05      Length of buffer
06      Low pointer to buffer
07      High pointer to buffer
%


        ONPAGE[EOPage];
* Output microcode is notified at XOStart to initialize.
XONoWork:
XOStart:        XWTemp ← XWDisableOutput, CALL[XWOutState], AT [EOStartLoc];

* Idle state of output microcode.
XOBegin:        Pfetch4[XWCSB, XWCSB1, XWIndexHeader!];  * fetch pointer to first IOCB
        T ← XWNextIOCB;
XOCheck:        SKIP[ALU#0], XWIOCB ← T;  * ALU = 0 => no IOCB
        GOTO [XONoWork];

* check to see if we are sending the same packet twice
*       Pfetch1[XWCSB, XWTemp, XWIndexTransInt];
*       LU ← XWTemp;
*       SKIP[ALU=0];
*       BREAKPOINT;
*       NOP;

        CALL[XWBSetup];
        LU ← XWCount;
        SKIP [ALU>=0];
        GOTO [XOMark], T ← XWCompletion ← XWErrorZeroBuf;  * Buffer empty
        PFetch1[XWIOCB, XWTemp1, XWIndexTransInt!];
        SKIP [R>=0], XWTemp1 ← (LSH[XWTemp1,1])+1; *  R<0 => overflow
        GOTO [XOMark], T ← XWCompletion ← XWErrorCountOV;
        PStore1[XWIOCB, XWTemp1, XWIndexTransInt!];

* Compute countdown interval
* Use memory refresh address for random number.
XOCountdown:
        T ← Stkp;  * Save Stkp
        XWTemp3 ← pXWRandomReg;  * point to "random" register
        Stkp ← XWTemp3, XWTemp3 ← T; *** Expect interlock warning
*  Form new transmission interval mask, check if old has overflowed
        T ← CTASK;  * XOR with Task number to add randomization
        T ← (LDF [Stack, 4, 10]) XOR (T);  * Get bits 4-13B for random number
        XWTemp3 ← (XWTemp3) XOR (377C);  * Complement Stkp value
        Stkp ← XWTemp3;  * Restore stack-pointer
        XWTemp1 ← (RSH[XWTemp1,1]) AND (T);      * Mask random number with old value
* What is correct scale factor?
        GOTO [XOGO, ALU=0], XWTemp1 ← LSH[XWTemp1,2];
        T ← (LDF[XWTemp1, 7, 2])-1;
        XWTemp3 ← T, TASK;  * Save high part (minus 1) in XWTemp3
        XWTemp1 ← LDF[XWTemp1, 11, 7];
        XWTemp ← XWDisableOutput;
        OUTPUT [XWTemp, XWWriteState];
* XWTemp1 now has low 7 bits of random no.
* Start simple timer with low 7 bits of random number.
* Timer slot is EOTask.
        XWTemp4 ← XWTimerMask;  * Compute timer word
        T ← CTASK;  * Timer slot is same as output task no.
        T ← (XWTemp4) OR (T);
XOLoadTimer:
        XWTemp1 ← LSH[XWTemp1, 4];  * align data part
        XWTemp1 ← (XWTemp1) OR (T);  * OR in mask
        LoadTimer[XWTemp1], CALL[XWDally];
* Can't fiddle timers as last instruction in task
        GOTO[XWReturn]; * Driver poked us while we were waiting

* Timer has expired.
* Check if still more time to elapse before start of transmission (High part of random number >=0).
EOTimerDone:
        XWTemp1 ← 177C, AT [EOTimerDoneLoc];  * Set up maximum timer value
        GOTO [XOLoadTimer, R>=0], XWTemp3 ← (XWTemp3)-1;
*  Countdown interval is over
*   Start to output words to the hardware.  The hardware will start
*       the transmission to the Wire as soon as the buffer has
*       > 10 words in the hardware buffer, or the OutputEOP bit
*       is set (for a packet of less than 12 words)
        XWTemp ← XWEnableOutput, CALL[XWOutState];
XOGO:

IFE@[XWire, 0, COMCHAR@["], COMCHAR@[']];
*'  ---- This code for Ethernet boards.
        XWIndex ← (XWIndex)-(4C);  * Initialize displacement to -4
*'  ---- End of code for Ethernet boards.
```

```
*"  ---- This code for Xerox Wire boards. Send Preamble.
        T ← XWTemp ← XWPream;
        XWTemp ← (LSH[XWTemp,10]) OR (T);
        OUTPUT[XWTemp, XOData];
        XWIndex ← (XWIndex)-(4C);   * Let XWTemp be written
        XWTemp ← (XWTemp)+1;
        OUTPUT[XWTemp, XOData], CALL[XWDally];
*"  ---- End of code for Xerox Wire boards.
COMCHAR@[~];     *Reset back to ~

        CALL[XOLoop], XWCount ← (XWCount)-(4C);
*  Main loop-checks for end of buffer, then checks ATTEN for end-of-pkt.
XOLoop: GOTO [XOQuadEmpty, R<0], XWCount ← (XWCount)-(4C);
        $GoToIOATTEN [XOAbort], T ← (XWIndex) ← (XWIndex)+(4C);
        RETURN, IOFetch4[XWPtr, XOData];

* Normal exit from Output Loop is here
* Number of singles remaining in buffer = XWCount+8.
* Set up XWCount as (No. singles-1), and output singles.
XOQuadEmpty:     T ← XWIndex ← (XWIndex)+(4C);
        CALL[XOSingles], XWCount ← (XWCount)+(7C);
XOSingles:       GOTO [XOEnd, R<0],  XWCount ← (XWCount)-1;
        $SkipNOATTEN, PFetch1 [XWPtr, XWTemp];
        GOTO [XOAbort];
        T ← (ZERO)+(T)+ 1, XWTemp;       * Update T and Abort
        GOTO[XWDally], OUTPUT [XWTemp, XOData];  * OUTPUT Timing

*  END OF PACKET
XOEnd:
* Hack to generate lots of JAMs -- Just wait for an UnderRun
*       CALL[.+1];
*       NOP;
*       $SkipNOATTEN;
*       GOTO [XOAbort];
*       RETURN;

        XWTemp ← XWSetOutputEOP, CALL[XWOutState];       * Set OutputEOP bit
        NOP; * Alignment(?) XWire
*   Wait for end-of-packet wakeup
XOAbort:        INPUT [XWTemp, XWStatus];        * Read Status
        XWTemp ← (XWTemp) AND (IFE@[XWire, 0, XOSMask, XXOSMask]);       * Isolate OutStatus
        GOTO [XOPacketOK, ALU=0], LU ← (XWTemp) XOR (IFE@[XWire, 0, XOCollisionMask, XXOCollisionMask]);
* error reported by the hardware
        SKIP [ALU=0], T ← RSH[XWTemp, 10];
        XWCompletion ← XWErrorBadHWPkt, GOTO[XOMark];
        PFetch4[XWCSB, XWCSB1, XWIndexHeader!], GOTO[XOClear]; * Collision only

* GOOD PACKET status:
XOPacketOK:
        T ← XWCompletion ← XWGoodPacket;  * Mark packet good
XOMark: XWCompletion ← (XWCompletion) OR (T), CALL[XWStoreStatus];
        CALL[XWNotify];
XONextBuf:
        CALL[XWNext];  * Get next IOCB set up, and fall into collision.

* COLLISION, restart same packet.
XOClear:        XWTemp ← XWDisableOutput;
        OUTPUT [XWTemp, XWWriteState];
        XWTemp ← XWEnableOutput, CALL[XWOutState];
* Now check if there is a buffer.
        T ← XWNextIOCB, GOTO[XOCheck];
```

```
*   SUBROUTINES


* Get buffer descriptor pointed to in T,
*   and fix pointer (in XWPtr, XWPtrHi)  XWPtrHi goes from (0, x) to (x, x+1)
* Clears XWIndex
XWBSetup:
        PFetch4 [XWIOCB, XWIndex, XWIndexBuffer!];
        XWIndex ← 0C;
        T  ←  (XWPtrHi)+1;  * Fix up high pointer
* MUMBLE
        RETURN, XWPtrHi ← (LSH[XWPtrHi, 10]) OR (T);


* Various simple Load/Store routines that are handy for TASKing
*   T points to destination word

XWStoreStatus:  GOTO[XWReturn], PStore1[XWIOCB, XWCompletion, XWIndexCompletion!];
XWStoreIndex:   GOTO[XWReturn], PStore1[XWIOCB, XWIndex, XWIndexBuffer!];

XWStoreTemps:    RETURN, PStore4[XWPtr, XWTemp1];


* Chain to next IOCB and update pointer in CSB
* This could be merged with XWNotify,
*   but that would be too long between TASKs
XWNext:
        PFetch4[XWCSB, XWCSB1, XWIndexHeader!];
        LU ← XWCSB1;
        PFetch1[XWIOCB, XWNextIOCB, XWIndexNext!];
        LU ← XWNextIOCB;
        RETURN, PStore4[XWCSB, XWCSB1, XWIndexHeader!];

* Notify the driver.
XWNotify:
        PFetch1 [XWCSB, XWTemp, XWIndexWake!];  * Fetch wakeup bitmask
        LoadPage[0];  * 100000c means tasking return
        T ← (XWTemp) or (100000c), GOTOP[DoInt];

XWOutState:
        OUTPUT [XWTemp, XWWriteState];
XWDally:        NOP;  * Need at least 2 instructions after OUTPUT before task switch
XWReturn:       RETURN;

        END[XWTask];
```