ARC Development System User's Guide

Manual Edition 1.0.1
Documents ARC version 1.4
Kernel version 0.986
May 6, 1996

Anne Wright, Randy Sargent, and Carl Witty
Newton Research Labs
Evergreen Park
16504 NE 37th St. #X2073
Redmond, WA 98052
Voice: (206) 869-4988
Fax: (206) 882-2043
E-mail: arc@newtonlabs.com
http://www.newtonlabs.com

1 Installing ARC on your system

This section is intended to guide an administrator through the process of setting up a new ARC installation. If your project already has an ARC installation and you want to start using it, see Chapter 2 [Getting started using ARC], page 7 instead.

1.1 Create a main directory for ARC on your system

The first step in installing ARC on your system is to decide on a place in your directory structure for the arc binaries and libraries.

- If you are installing on a stand alone machine or a small network, we recommend making the directory '`/usr/local/arc`' and using that as the primary directory for ARC.
- If you are on a large network and ARC is being used by multiple people in your project, it may be more appropriate to install arc in your project's main directory. This might be something like '`/projects/walker/arc`'.
- If you are the only person using ARC, you may want to put the main ARC directory in your home directory (as in '`~/arc`').

These suggestions are more for aesthetic purposes than functional ones. So long as your path is set up properly, proper functioning of the software is not affected by where the main ARC directory is located. However, it is good practice to keep software packages which are used by multiple people in a more central location than someone's home directory. On unix systems, '`/usr/local`' is a standard place to put such packages, so that will be the default in examples in the rest of the manual.

Once you have decided where the main '`arc`' directory should be, make the directory (ex. `mkdir /usr/local/arc`) and give it appropriate permissions (ex. `chmod g+rw /usr/local/arc`). This will be the directory in which subdirectories for ARC binaries, libraries, etc. will be placed.

1.2 Decide on which host and target packages you need

ARC contains pieces which depend on both the type of host (the computer you sit at and run compiles on) and the type of the target (the processor you are ultimately trying to program).

1.2.1 Host Platforms

Currently available host platforms are:

- Sparc running SunOS
- IBM PC or compatible running Linux
- IBM RS6000 running AUX
- HP700 running HPUX
- NeXT running Mach

If your host platform does not appear on the list, please contact Newton Labs (see Appendix A [Reaching Newton Labs], page 89). So long as it is possible to build a gcc cross compiler for your host platform, it is possible for us to build ARC for your needs. In general, any machine running UNIX will probably be acceptable as a host platform. Windows NT may work as a host platform soon. Macintosh, DOS, and Windows 3 are not likely to work as host platforms.

1.2.2 Target Platforms

Currently supported target platforms are the following:

- Motorola BCC 68332 board
- Vesta SBC332 board
- Onset Tattletale board

If you have another board based on a processor in the Motorola CPU32 family, it should be possible for you to use ARC. Contact Newton Labs (see Appendix A [Reaching Newton Labs], page 89) for details.

We have also started working on a port to 386, 486, and Pentium processors, but this system is not yet ready for distribution.

1.3 Copy the software to your site

ARC is distributed over the internet and can be copied using any web browser. The programs available over the network are initially disabled, and require a license key which must be purchased from us before the software will run.

Use your web browser to go to the arc home page at '`http://www.newtonlabs.com/arc`'. Links from that page will get you to the arc download site, as well as online version of this manual and other related information. Follow the link to the ARC download page.

Select an appropriate package for each of the host and target, as discussed above. Instructions on the download page will help you to get the most current version. You may also want to check this site periodically for upgrades. Your original key will work with such upgrades, and you can follow these same instructions to install upgrades as well as the initial installation.

Click on the appropriate packages, and tell the browser to copy them to your main '`arc`' directory see Section 1.1 [Create a main directory for ARC on your system], page 1.

1.4 Extract the software from the tar archive files

The packages for the unix versions of ARC are compressed tar archive files. These should be extracted in the main '`arc`' directory (see Section 1.1 [Create a main directory for ARC on your system], page 1). They will expand to create separate directories for each host and target package.

In the following example, '`/usr/local/arc`' is the main '`arc`' directory. You start out in the directory with the compressed tar files for the sparc executables and the 68332 libraries, version 0.919. You move the files, uncompress them, untar them, and remove the archived versions. At the end of this example, you would have created and filled the directories '`/usr/local/arc/sparcbin`' and '`/usr/local/arc/332libs-0.919`'.

```
mv arc_sparcbin-2.0.tar.Z /usr/local/arc
mv arc_332libs-0.919.tar.Z /usr/local/arc
cd /usr/local/arc
uncompress *.Z
tar xvpf arc_sparcbin-2.0.tar
tar xvpf arc_332libs-0.919.tar
rm arc_sparcbin-2.0.tar
rm arc_332libs-0.919.tar
```

1.5 Install the license key

Once you have installed the packages, you need to install the license key before the software is fully enabled. This procedure requires that you have write access in the host tools directory (ie. '`sparcbin`', '`linuxbin`', etc), and that you use a machine of the appropriate architecture. For example, it is not possible to install the key for the sparc version while running on a linux machine – you would have to be on a sparc. If you do not have write permission in the directory, you may need to contact someone who does before continuing.

License keys have two parts: a string specifying the person or company which '`arc`' is licensed to, and a numeric sequence containing the actual key. Both parts must be entered correctly for the software to be fully enabled.

Go into the host tools directory and type `arc -config`. This will put you in arc configuration mode. There are many configuration parameters which may be set in this mode, including setting the license key information.

Press return until you are prompted with the line:

```
Person, Company ARC is licensed to:
```

Backspace to the start of the line and type the text part of the key. Then press return. You will next be prompted with the line:

```
License key:
```

Again, backspace to the start of the line and type the numeric part of the key. This will be a string of numbers separated by hyphens.

Other configuration options may be safely ignored for now. Hit return at each and they will retain their default values.

Run '`arc`' again, but this time without the `-config` flag. If the key was installed properly, it will say:

```
Single user license for: Joe the Robot
License key: XXXXX-XXXXX-XXXXX-X

Port /dev/cua0, baud 38400
```

```
    Terminal port # 9041
Term socket = <fd 4>
arc>
```

If the license was not installed properly, or was not a valid 'arc' licence, it will say something like:

```
Missing license key.  Please contact Newton Research Labs for key, and install
keys by typing "arc -config" from arc's home directory.
```

Then it will exit. If this happens, check that you typed both parts of the key accurately. If the problem persists, contact arc@newtonlabs.com.

1.6 Set up the default kernel

It is important to specify which kernel to compile against before trying to use the ARC system. This specification can be made at several different levels:

- A command-line option can be passed to `arcc`. see Section 4.2 [Using arcc], page 31.
- A user-specific default can be set up each user's '`~/.arcrc`' file. see Section 4.1.3 [Specifying your personal default kernel], page 31
- A kernel can be symbolically linked to '`default`' in a subdirectory of the main '`arc`' directory.

The options in this list are looked for and used in order. For example, if a command-line option is specified, the other options are not pursued.

In order for the ARC Development system to work for new users without individual customization, it is best to set up a '`default`' symbolic link to the most likely kernel in the file system.

Kernels are of the form:

'`arc/332libs-x.xxx/vestaboot.xxx`'
: A Motorola 68332 kernel which will run on a Vesta SBC332 board

'`arc/332libs-x.xxx/ttaleboot.xxx`'
: A Motorola 68332 kernel which will run on an Onset Tattletale 68332 board

'arc/386libs-x.xxx/pckernel.xxx'
: An Intel 386, 468, Pentium kernel which will run on an IBM PC (not yet in distribution).

To discover which version a given board is running, look on the ROMs, or connect the board to `arc`, hit reset, and look at the boot line. It should have the version number in the boot message.

```
ARC Multitasking 68332 Kernel Version 0.919
Copyright Anne Wright 1993

Compiled May  3 1995 at 13:09:12

vestaboot.919>
```

In this example, the board is running version 0.919 of the 68332 kernel for Vesta boards, so you should do the following to set up the appropriate default:

```
cd /usr/local/arc/332libs-0.919
ln -s vestaboot.919 default
```

Be sure to only have one '`default`' symbolic link in any of the subdirectories of the main '`arc`' directory at a time. To find out if there are any other defaults, you can do the following:

```
cd /usr/local/arc
find . -name "default" -print
```

This will print out all the files named default. Delete all but one.

1.7 Delete old versions of the libraries

To avoid problems of version skew, each update of the kernel, libraries, and header files are distributed to install in a different directory. This allows some people using a given installation of ARC to use old versions, while others upgrade to newer versions. However, if you are sure that no one is using particular old versions, delete them. The newer versions are all autonomous of the old ones.

2 Getting started using ARC

This chapter assumes that the ARC software has already been installed on your system, though you may not have used it yet. If you know that ARC has not been installed on your system, see Chapter 1 [Installing ARC on your system], page 1. If you are not sure, or if you do not know the location of the main ARC directory in your system, you should contact the person at your site in charge of the ARC installation before continuing.

This section is mainly intended to bootstrap new users, but may be useful to even experienced users when they need to break in a new board.

2.1 Set up your environment to use ARC

You should have, somewhere in your system, a directory called "arc". This is often located in /usr/local/arc, but could be elsewhere, depending on how your system is set up. If you don't know where this directory is, contact the person at your site in charge of the ARC installation before continuing.

Once you've found where the "arc" directory is, you'll need to figure out where the executable files live. The executables will be in a subdirectory of arc, but the exact name depends on your host system architecture. Here are some examples:

SPARC running SunOS
Binaries directory is '`arc/sparcbin`'

Linux Binaries directory is '`arc/linuxbin`'

RS/6000 Binaries directory is '`arc/rsbin`'

Once you've found the directories with the executables, you'll need to add this directory to your path. If you use csh or tcsh, you probably want to add something like the following to the end of your ".cshrc" file:

```
set path=($path /usr/local/arc/sparcbin)
```

(You should replace '`/usr/local/arc/sparcbin`', with the path to the binary directory you determined above.)

If you are using '`csh`' or '`tcsh`' as your shell, you can load the changes by typing '`source ~/.cshrc`'. This will only affect the particular shell you execute it in. If you log out and back in, the path modifications should be executed for all of your shells.

If you are using a shell other than '`csh`' or '`tcsh`', you may need to do something different to extend your path. See the manual page on your shell for details.

2.2 Set up to use your serial port

Before starting ARC, you will need to know the name of the serial port you intend to use. The device names vary from system to system. Often, a serial port is set up to not give read and write permissions to standard users. Occasionally, a serial port is set up to run "getty", which means the system thinks a serial terminal (such as a VT100) is connected, and that users will wish to log in. If in doubt, you should talk to your local system administrator to help you get started using the serial port.

Here are some typical serial port names, but realize that names sometimes vary even between different machines of the same architecture, so the following are suggestions for names to try, rather than the final answers:

- Linux: '`/dev/ttyS0`', '`/dev/ttyS1`', ...
- SPARC running SunOS: '`/dev/ttya`', '`/dev/ttyb`'
- RS/6000: '`/dev/tty0`'
- NeXTStation: '`/dev/ttya`', '`/dev/ttyb`'
- DECstation and HP Snake: '`/dev/tty00`', '`/dev/tty01`'

You may want to confirm that the serial port works before you try to use ARC to talk to a processor board, as it is good to keep the number of unknowns to one at a time. One way to do this is to hook the serial port on your host machine to a serial source that is known to work (for instance, a modem or another serial port) and run `kermit`.

It is critical to get this step right before you continue. If you have not used serial ports on a unix machine before, I highly recommend you find someone more experienced to help you.

2.3 Connect a serial cable to your processor board

The issue of serial cabling is very complex. This section only handles specific issues relating to using 68332 boards. For more general information, see Appendix B [Building Serial Cables], page 91.

2.3.1 Motorola BCC and Vesta boards

The serial port on both the Motorola BCC and Vesta boards is a piece of male header with the following pinout:

```
|       |       |       |
|   3   |   2   |   1   |
|________________________
 332RxD  Ground  332TxD
```

On the Motororola BCC this connector is P4, on the Vesta it is J5 (see Section 7.1 [Vesta Board Diagram], page 84). Pin 1 is labelled and the same on both boards.

Many people make a serial cable for these boards by soldering one end of a piece of 3-conductor wire to a piece of female header, and the other end to a DB25 or DB9 cable. If you do this, I highly recommend that you mark pin 1 on both the cable and the board with some paint.

Motorola BCC or Vesta to RJ-11 converter

A better solution is to make a converter from the 3-pin serial header to a female phone jack, and then use the instructions in (see Appendix B [Building Serial Cables], page 91). I also recommend that you replace the male header with a 3-pin keyed header and build a 3-pin header to RJ11 adapter with the following parts avaliable from Digikey:

```
WM4301-ND       Keyed 3-pin male header connector (solder to board)
H9011-ND        RJ-11 6-4 Jack with 6" leads (goes to housing)
WM2001-ND       Keyed 3-pin female housing (goes to RJ11 jack)
WM2200-ND       Crimp terminals for housing (crimp on wires, put in housing)
```

- Solder the red and green wires together and crimp a terminal on them. This will be your ground connection. Put this terminal in the center of the housing.

- Crimp a terminal on the black wire. Place the black terminal in the correct hole to connect to pin 1 on the board. This wire is transmit from the 332, and receive to the host.
- Crimp a terminal on the yellow wire. Place the yellow terminal in the correct hole to connect to pin 3 on the board. This wire is receive to the 332, and transmit from the host.

How to tell if you got your Motorola or Vesta cable right

When plugged into the host computer (your workstation) and not the 332 you will get about -10V between pins 2 and 3 on the connector to the 332. Check this now. If you got it backwards and have no voltage between pins 2 and 3, but -10V between pins 1 and 2, you can either change the cable or install a null modem.

2.3.2 TattleTale 332 board

The tattletale board serial ports use stereo headphone jacks, and you should have a stero-jack to DB9 cable or stereo jack to 8-pin mini DIN cable that came with the board.

8-pin mini DIN to sereo jack cable

If you are using a NeXT or some Sparcs, you can get a mini-jack to DIN cable from Onset that I expect would work, though I haven't actually tried one. If you are using a Mac, you are out of luck because none of the arc development system currently works on a Mac. If you are using any other machine, I recommend getting the DB9 cable instead.

DB9 to stereo jack cable

The pinout for the DB9 end of the cable is the standard IBM PC pinout:

```
     5 4 3 2 1     This is the pinout as you look at the DB9 end
      9 8 7 6      of the cable from the board

  Pin 2: Host serial receive  (332 board serial transmit)
  Pin 3: Host serial transmit (332 board serial receive)
  Pin 5: Ground
```

If you are using an IBM PC with a DB9 port, you are in luck. Otherwise, follow the instructions in (see Appendix B [Building Serial Cables], page 91) to convert the DB9 to the standard RJ11 connector just as you would if the DB9 were on a host instead of to the 332 board (the cabling scheme is symmetric).

How to tell if you got your Tattletale cable right

When plugged into the host computer (your workstation) and not the 332 you will get about -10V to -5V between pins 3 and 5 on the DB9. Check this now. If you got it backwards and have no voltage between pins 3 and 5, but -10V to -5V between pins 2 and 5 instead swap pins 2 and 3 until this test works.

2.4 Check that you can run the ARC serial interaction program

You should have a serial cable hooked up between your host computer and your board now.

To run arc, type:

```
arc -port /dev/your-serial-port
```

Arc should give you a startup message and then give you the "arc>" prompt.

If this doesn't work, check that your path includes the correct arc binary directory for your machine type. See (see Section 2.1 [Set up your environment to use ARC], page 7) for details.

2.5 Add a reset button

Vesta Reset button

On a Vesta board you can put a normally open pushbutton switch across the two pins of J4 to make a reset button (see Section 7.1 [Vesta Board Diagram], page 84). I recommend soldering a few inches of wire to a pushbutton switch on one end, and a two-long piece of female header on the other. Attach this to J4 (on the same end as the serial ports, towards the other side).

Motorola BCC Reset button.

If you are using the platform board, then you already have a reset button. If not, add the following circuit (if you can):

```
 +5V
 ----
  |
  \
  /
  \ 10K Resistor
  /
  |---------- /RESET signal to pin 57 on connector P1
  /
 /  Reset button (Normally open)
  |
  |
 ----
Ground
```

I've never figured out a good way to do this, other than to stack the BCC on top of a wire-wrap board with a reset button on it.

Tattletale Reset and Download buttons

On a Tattletale, there are two hardware additions which should be performed on the Tattletale expansion board regardless of your application: a RESET button, and a DOWNLOAD button. Both of these should be normally open buttons connected as shown below. The RESET button will pull the MCLR line low when pressed and reset the board. The DOWNLOAD button, if pulled low during RESET will allow code to be downloaded to EEPROM.

```
RESET Button                  DOWNLOAD Button
  +5V                              +5V
  ----                             ----
   |                                |
   \                                \
   /                                /
   \ 10K Resistor                   \ 10K Resistor
   /                                /
   |---------- /MCLR                |---------- /IRQ3
   /                                /
  /  Reset button                  /  Download button
   | (Normally open)                | (Normally open)
```

```
         |                                |
        ----                             ----
       Ground                           Ground
```

Plug the board in and test that the MCLR line is pulled low when you press the RESET button, and is high otherwise. Similarly test that the IRQ3 line is pulled low when you press the DOWNLOAD button (it should also cause a little red LED on the bottom of the board to light), and high otherwise.

2.6 Add power to your board

2.6.1 Motorola BCC and Vesta Power Connection

The Motorola BCC and Vesta boards require a regulated 5V power supply.

On a Vesta board, the power connector is J6 (see Section 7.1 [Vesta Board Diagram], page 84). Pin 1 is ground and pin 2 is +5V.

On a Motorola BCC there is no connector. You will either have to power it via another board (such as the Motorola platform board) or hook a power cable directly to one of the 64 pin connectors. The outermost pins (1,2,63, and 64) are all ground pins; the next innermost pins (3,4,62, and 61) are all power pins.

Be sure to get this right the first time. If you plug the power supply in backwards you may toast your board right now. None of these connectors provide any kind of diode protection and the components may not survive negative voltage being applied. Immediately check that there is +5V across the power and ground connectors on the 64 pin header as described above.

2.6.2 Tattletale Power Connection

Tattletale boards have a 5V regulator on board. Therefore you can use anything from 6V to 12V to power the board. However, in order to program the EEPROM, you should use a 12V DC power supply.

On the Tattletale expansion boards, there is a coax power connector, which is a common connector on the end of AC/DC transformers.

```
************************ WARNING **********************************
```

Although using coax power connectors is common, the polarity is not standard. Devices and supplies come in two different flavors – tip positive and tip negative. The Tattletale expects TIP NEGATIVE DC power. The symbol for the one you want looks like this:

```
                          ----
                         /
      (+)   -----------|  O-------  (-)
                         \----
```

The polarity of the outside part is on the left attached to the arc, and the polarity of the tip is on the right attached to the dot in the middle. Most AC/DC converters are marked similarly to this. (Though even if it is labeled I still recommend double-checking with a multimeter).

2.7 Make sure your board boots and talks to ARC

To test your target board and its connection to your host computer, press the board's RESET button, and you should see the board print a startup message onto the main ARC screen.

The startup message should look something like this:

```
Anarchboot Multitasking 68332 Kernel Version 0.919
Copyright Anne Wright 1993

Compiled May  3 1995 at 13:09:12

Checking for reset run programs
Control-d to inhibit reset run...10...9...8...7...6...5...4...3...2...1...
Running program vision as process 99
vestaboot.919>
```

If you do not get this, confirm that power is applied correctly to your board, confirm that the board is hooked up to the correct serial port, and confirm that your serial cable is contructed correctly. Do not try to go on until you get this step to work.

2.8 Load a program to your board using ARC

By now you should have ARC running on your host machine communincating with your processor board. You are now ready to load a program.

If your system has EEPROM (all Tattletale boards and modified Vesta boards), you have the option of loading programs to EEPROM as well as RAM. This will allow programs to persist reliably even when the board is powered off. These examples use the `ramload` command which only loads to RAM. They would also work for loading to EEPROM by giving identical commands to te `romload` command. See Section 3.4.1 [Loading user programs and libraries in ARC], page 22 for more information.

To compile and load a program into arc, type:

```
arc> ramload progname.c
```

This command calls the `arcc` program to deal with the compilation, then downloads the resulting a.out file. If the compilation fails, double check that your arc directory see Chapter 1 [Installing ARC on your system], page 1 and environment see Section 2.1 [Set up your environment to use ARC], page 7 are set up properly.

If the compilation succeeds but it does not download properly to the board, go back and double check that you can talk to '`arc`' successfully see Section 2.7 [Make sure your board boots and talks to ARC], page 14.

Once the download is completed successfully, You can then try running your program by typing:

```
arc> run progname
```

If the program is configured to run when reset is pressed, you can also run it simply by pressing the board's reset button. By default, programs will run on reset. See (see Section 4.3 [Changing program startup parameters], page 33) for information.

Congratulations! You have now worked through all the steps required to use the ARC Development system.

3 Using the ARC serial interaction program

The ARC serial interaction program, `arc`, is the link between the host computer and the target embedded system. After a program has been written and compiled on a host system, `arc` is used to download the program and interact with it over the serial port.

In normal use, `arc` is the main program a user of the ARC development system will interact with. Many other programs are involved in process of compiling and executing code, but they mostly work in the background. If you specify a `.c` source, or `.lis` list file to any of the `arc` load commands, `arc` will automatically invoke `arcc` to compile the program before download. In the case of compile errors, the errors are reported and the download aborted.

`arc` also provides an interpreted C interaction environment with the ability to access and modify global variables and call functions on the board which is very helpful for debugging. After reading in the symbols as a side effect of a load or file command, you can type most valid C expressions at the `arc>` prompt and they will execute interactively on the board.

3.1 Overveiew

The `arc` program is both the most central part of the ARC Development System, and the most confusing. To explain what `arc` is, it is perhaps best to go back to the beginning and explain what `arc` used to be.

The program which has evolved into `arc` started life as `term`, a serial-port terminal program similar to `kermit`, but much simpler, written by Randy Sargent. All that `term` did was echo characters from the serial port to the screen, and from the keyboard to the serial port. Features were sometimes added to `term` to do things like download S-records over the serial port when particular escape characters came from the keyboard. `term` was an excellent tool for dealing with serial interaction to single-threaded processor boards, and stayed largely unchanged for years.

Then I wrote a multitasker for the 68332, and wanted to be able to interact with the various processes, each over an individual serial stream. `term` was too limited for this task, so Randy made a new version which could deal with input from and output to multiple streams, each in its own X window. The streams were multiplexed onto a common serial port using a simple stream-break protocol. This new program was called `multiterm`.

Later, we decided to try to interface a process on the board to the `gdb` remote protocol to allow symbolic debugging. We wanted to be able to interact with the board while at the same time debugging it using the `gdb` remote protocol. We added to `multiterm` the ability to create bidirectional streams to other programs over sockets, and multiplex those streams over the serial port to the board as well.

Using `multiterm` and `gdb` connected together via sockets like this worked, but it was very confusing to use. We also had problems because `gdb` was very difficult to modify, and it did not interact well with multitasking. So we set out to integrate the functionality of `multiterm` and `gdb` together into a single, easy-to-use program called `arc`.

Like `multiterm`, `arc` is at heart a multi-streamed serial interaction program. However, because `arc` is also used to load programs and perform many debugging features, it also has a prompt which you interact with as if it were a shell. Balancing these features is somewhat difficult. It would be much easier to understand what is going on if it were not possible to interact directly to different streams and processes, but I believe that the benefits outweigh the problems.

3.2 Using ARC

In using `arc`, there are two modes, prompt and passthrough, and two types of windows, the root window and process windows.

3.2.1 Prompt vs. passthrough mode

In prompt mode, you type commands at the `arc>` prompt and `arc` interprets the command. As a result of commands, serial data may be exchanged with the board, or it may not. Any characters which are received from the board are still printed to the screen, even though you are in prompt mode, but characters you type are not sent directly to the board.

In passthrough mode, each character you type is sent straight through to a serial stream to the board, and each character you see printed came from the board.

In any window, you can switch between prompt mode and passthrough mode by typing control-x (hold down control while typing x – usually denoted as ^x).

3.2.2 ARC root window

The root window is the part of the `arc` interaction which remains in the window in which you invoked `arc`. The root window actually interacts with three separate processes on the board and multiplexes their streams all into the same window. The three processes are:

- The console process, which provides low-level debugging monitor features; status reports on memory usage, multitasking, etc.; modification of persistent variables (see Section 6.5 [Persistents], page 52); and allows you to run programs on the board. For more information about the console commands, see Section 5.4 [User console], page 40.
- The debugging process, which handles debugging interaction to `arc` in the background whenever a program is downloaded, a variable is accessed, or a function is called in the `arc` interactive debugging mode
- The dummy process, which is the default context in which interactive function calls are executed. When not executing interactive function calls, this process is idle.

Certain `arc` commands executed at the prompt in the root window will cause the root window to enter passthrough mode until some completion sequence is received from the board. For example, executing any console commands will place the root window in passthrough mode until the console command completes, and executing interactive functions will place the root window in passthrough mode until the end-of-function message is received from the debugging process.

At any time, you can toggle between passthrough and prompt mode by typing ^x (control-x), but be aware that in the current version, entering passthrough mode manually may result in connecting you to the dummy process, which does not respond when it is not in the process of executing code. If you hit return and do not get a prompt, you are probably in passthrough mode, and you should type ^x to return to prompt mode.

Additionally, if you are executing an interactive function call, you can type ^c (control-c) to halt the dummy process so you can debug it. This will also have the effect of placing you in prompt mode.

3.2.3 Process windows

When `arc` receives serial from the board on a stream other than the three system streams (see Section 3.2.2 [ARC root window], page 19), it brings up an xterm communicating with that stream. By default, when a process window is brought up, it is in passthrough mode.

Because `arc` and the board are not tightly synchronized (a decision made for runtime performance reasons), `arc` does not necessarily know that a stream has been created until it says something. If you want to bring up a window to interact with a stream that has not yet said anything, use the `window` command from the `arc>` prompt, either from the root window or in prompt mode in any window (see Section 3.3 [ARC commands], page 20).

If you want to debug a process, go to its window (if it doesn't have one, create it) and type ^c (control-c). This will send a message to the board to halt the process associated with the window. After that, you can use the `where` command in the halted process' window for a stack backtrace, or the `cont` command to continue the process and return to passthrough mode.

At any time, you can toggle between passthrough and prompt mode manually by typing ^x (control-x).

3.3 ARC commands

Commands recognized by `arc` as console commands are passed through to the console process on the board. For details on console commands, see Section 5.4 [User console], page 40.

Additionally the following arc-specific commands can be typed at the `arc>` prompt:

'`ramload <filename> [-addr <start-addr>] [-nosyms] [<arcc args>]`'

'`romload <filename> [-addr <start-addr>] [-nosyms] [<arcc args>]`'

'`kernelload <filename>`'

'`bsload <bootstrap> <loader>`'

These commands are all used to download programs to the board. Different commands are appropriate for different uses. See Section 3.4 [Loading files in ARC], page 21 for details.

'`file <filename>`'

Read the symbol table of the specified executable file, but don't download it to the board. Use this if you want to debug a program which is already on the board. Be careful using this, as there is no way to for `arc` to check that the symbol table you are reading in is the same as the program which is resident on the board. For this reason, it is generally safer to use the symbol table which is loaded as a side effect of loading the program.

'`hostbaud <baudrate>`'

Change the host baud rate unilaterally. Default value for the baud rate is 38400.

'baud <baudrate>'
: Change the board and the host baud rate. The board and host must already be at the same baud rate for this to work.

'window <pid>'
: Create new window talking to the process with the specified process id. Use the ps console command to determine the correct pid to use.

'cd <dir>'
: Change current working directory to on the host to <dir>.

'pwd'
: Print the current working directory on the host.

'cont'
: Continue a process being debugged. If executed in a process window, it continues the process associated with that window. In the root window, it will continue the default interaction process if it is active.

'where'
: Print the stack backtrace of a process being debugged. This should be executed at the prompt in a process window in which the process has been broken (either due to a crash or by typing control-c). See Section 3.2.3 [Process windows], page 19 for more information.

'log <logfile>'
: Start logging the window log was executed in to <logfile>.

'endlog'
: Turn off logging in the window endlog was executed in.

'help'
: Display this list for the version of arc you are using.

3.4 Loading files in ARC

There are three different kinds of files which can be downloaded to the board.

User programs and libraries
: These are programs compiled using 'arcc'. They are linked against kernel, but are placed elsewhere in memory from kernel. Therefore kernel is capable of loading and unloading this kind of program.

Kernel
: The kernel contains the operating system and libraries which are shared by all user libraries and programs. See Chapter 5 [Kernel], page 35 for more information. Most programs are downloaded by arc sending download messages to the kernel. Unlike user programs, most of the kernel runs in ROM, and you can only download the kernel on 332 systems with EEPROM. This includes only Tattletale boards and modified Vesta boards. Because the kernel cannot load over top of itself, so you cannot upgrade the kernel using the regular load commands.

Bootstrap Starting with version 0.980 of the kernel, a bootstrap program is available for EEPROM equipped 68332 systems (Tattletale and modified Vesta boards) which is capable of booting the system and downloading new kernels. A special `arc` command allows downloading new bootstraps.

These different kinds of files require different programming strategies, and therefore have different `arc` commands.

3.4.1 Loading user programs and libraries in ARC

```
ramload <filename> [-addr <start-addr>] [-nosyms] [<arcc args>]
romload <filename> [-addr <start-addr>] [-nosyms] [<arcc args>]
```

The `ramload` and `romload` commands in arc allow loading user programs and libraries. `ramload` downloads the program only to RAM, which is volatile memory and may not persist across power cycles. `romload` downloads the program both to non-volatile EEPROM and to RAM. When the board starts after a power cycle, it will find the programs stored in EEPROM aand restore them automatically.

While `ramload` works for any board, `romload` only works for boards equipped with EEPROM. This includes all Tattletale 332 boards, and specially modified Vesta 332 boards.

Both of these load commands are capable of loading pre-compiled executables, or calling the compiler and downloading the result. If the file specified is an a.out format executable or S-record file ('`.s19`' extension), it is downloaded directly (without recompilation).

If the file specified is a `.c` source, or `.lis` list file, `arcc` will be invoked to compile the program before download. In this case, any extra arguments on the command line are assumed to be `arcc` flags and are passed through to the compilation. See Chapter 4 [Compiling programs under ARC], page 29 for more information about `arcc` flags. In the case of compile errors, the errors are reported and the download is aborted.

Programs are not relocatable once they have been linked (the final stage of compilation). Therefore, a given program can only be executed when it occupies the particular place in memory at which it was linked. The `ramload` command reads this location from the executable and downloads it there by default. The `romload` command, on the other hand, is not constrained to loading to a particular address. The copy of the program in EEPROM is there for nonvolatile storage, and must be copied to RAM before being executed. Therefore `arc` will communicate with the board to find a free block in EEPROM and download the program there. After download the EEPROM, the program is automatically copied to the appropriate location in RAM and can be executed immediately.

The optional `-nosyms` flag will prevent the symbol table from being loaded. Symbol tables are somewhat large and slow to load, so this option allows you to speed downloads by skipping this step when you do not intend to debug the program right away.

The optional `-addr` flag followed by an address is used to specify the memory address to download the program to. In general, you should not need to use this argument, as both load commands should come up with reasonable addresses on their own. You can override the automatic download address with the `-addr` flag. This is generally pointless in the case of `ramload`, but it is sometimes useful to override the placement selected by `romload` with your own choice. Use the `memmap` and `memfree` console commands to help you decide what locations would be good (see Section 5.4.3 [Viewing memory usage], page 42).

Note that he `-addr` argument does not affect compilation, only download. See Section 4.2 [Using arcc], page 31 for information on how to set the location at compile time. In particular, look at the the CODE_ADDR environment variable and commandline option.

3.4.2 Loading multiple C files in ARC

There are three different ways to load a program consisting of multiple C files using ARC, listed in order of increasing difficulty and increasing functionality:

3.4.2.1 Specifying multiple C files in load line in ARC

The simplest way is to specify multiple C files on the same line when you use the '`ramload`' or '`romload`' command in ARC. (Both '`ramload`' and '`romload`' work the same way with respect to this feature, so the use of `ramload` in these examples is arbitrary).

```
arc> ramload file1.c file2.c
```

Note that including multiple files in the same `ramload` command as shown above is very different than specifying them in separate commands as below:

```
arc> ramload file1.c
```

```
arc> ramload file2.c
```

The first example will result in '`file1.c`' and '`file2.c`' being linked together into the same executable. The second example will result in '`file1.c`' and '`file2.c`' being compiled into separate executables. Unless the CODE_ADDR `arcc` option is used to link the two files to non-overlapping addresses, '`file2`' will be written over `file1` and '`file1`' will be lost.

Note that while in other respects the behavior of `ramload` and `romload` are similar, at the current time the above example would have a different result if `romload` were used. In that case, both programs would be downloaded to EEPROM and the overlap would not be noticed until the programs were restored. This aspect of `romload` behavior is still undergoing specification, so just be aware that this may lead to unexpected results.

3.4.2.2 Using list files

An even better way of loading multiple files is to create a list file. This is a file with a '`.lis`' extension which lists the files you want to load together, as well as relevant compile flags and options. For example:

List file prog.c:

```
file1.c
file2.c
CODE_ADDR=0x108000
KERNEL=vestaboot.986
-O2
```

In this example, the first two lines contain source files to link together. The next two lines contain arguments to `arcc` which affect how the program is linked (see Section 4.2 [Using arcc], page 31), and the last line contains a compiler flag which `arcc` passes along to `gcc`.

To load the list file type:

```
arc>  ramload prog.lis
```

This will compile '`file1.c`' and '`file2.c`' together into an executable named prog and download that program to the board. (You could also have used '`romload`' to also load a copy to EEPROM).

3.4.2.3 Using Makefiles

Using 'make' and Makefiles is very complex, and a full explanation is outside the scope of this document.

The basic idea behind using 'make' is that you set up rules in a file (usually called 'Makefile') in your project directory that describe how to convert source files into object files. When you run 'make', it checks if any source files are more current than their associated object files. If it finds any more current source files, it executes the specified commands to convert the source files into the object files.

At this point, there is no support inside arc for calling make automatically. Therefore, if you are using makefiles it is necessary to compile the file in a different window, then download the resulting executable using one of the load commands described above.

An example of using 'make' with the ARC development system to compile source files 'foo.c' and 'bar.c' into the executable file 'baz' is as follows:

The following text is in a file named 'Makefile' in the same directory as 'foo.c' and 'bar.c'

```
all: baz

baz: foo.c bar.c
        arcc foo.c bar.c -o baz
```

To compile baz, you can then type 'make' in that directory.

Makefiles are good in that they check modification times and only do the compiles which are necessary. However, they are very hard to use. Unless you are already used to using Makefiles, I recommend that you use list files instead (see Section 3.4.2.2 [Using List files], page 24).

3.4.3 Loading a new kernel using ARC

```
kernelload <filename>
```

NOTE: This only works on systems with EEPROM (Tattletale and modified Vesta 332 boards).

The `kernelload` command uses the bootstrap to download a new kernel. You cannot use the `kernelload` command until you have installed a bootstrap. If a bootstrap is installed, the board will say something of the following form at 38400 baud at boot time:

```
BOOTSTRAP 1.0
```

If your board does not say this, then you need to install a bootstrap. See Section 3.4.4 [Loading a new bootstrap using ARC], page 27 for details.

The filename provided to kernelload can be either an a.out executable file or an '`.s19`' file. Generally this would be the same file as you would link your programs against, for example '`vestaboot.986`'. See Section 4.1.1 [Choosing the kernel which is right for you], page 29 for details.

Once you know that you have a bootstrap installed and know which kernel you want to load, run `arc` at 38400 baud. For example, `arc -port <port> -baud 38400`.

Definitely do not use the `-saphira` or `-saphira_stop` options as these will cause the baud rate to be set at 9600, which will not work for this application. Then execute the `kernelload` command. It will prompt you to ask whether you know what your are doing:

```
arc> kernelload /usr/local/arc/332libs-0.986/vestaboot.986
This command only works if you've installed a bootstrap
in your eeprom.  If you currently are using a pre-0.981
kernel and haven't loaded a bootstrap with 'bsload', please
locate instructions for loading the bootstrap first
Do you wish to download a new kernel at this time? (y/n)
```

Answer 'y' to continue. It will prompt you to turn off the processor board and press return to continue.

```
Please turn the processor board back on.
Download will start when arc has synchronized with the bootstrap
arc>
```

When you turn the board back on, the kernel download will begin. (The reason for turning the board off and on agan is that the bootstrap is only active at boot time, and waits for a second to hear whether `arc` has a program to download to it. After you have turned the board off, arc will start sending download commands so that the board will hear it immediately when it starts back up.)

The download takes a while, during which the bootstrap will print out its progress frequently. Wait until it is done, the turn the board off and back on again. With any luck, it will then be running a new kernel.

Be sure to relink and reload all of your programs after changing the kernel you are using, as all programs link against the kernel and therefore depend upon it. As part of the compilation process `arcc` includes version information for dependency checking, so the board should catch any version problems and refuse to run the outdated programs.

3.4.4 Loading a new bootstrap using ARC

```
bsload <bootstrap> <loader>
```

NOTE: This only works on systems with EEPROM (Tattletale and modified Vesta 332 boards).

It should only be on rare occasions that you would need to download a new bootstrap. It should not change very often. It does not have any dependencies on the kernel version, etc. However, it is necessary to do this once if you have pre-bootstrap EEPROMS, and it may be necessary to upgrade the bootstrap at some point.

The bootstrap and loader programs should be in the main kernel directory (for example, '`/usr/local/arc/332libs-0.986`'). You will need to cd into that directory (or specify the full paths) to download the bootstrap. The current versions are called '`vbs-1.1`' (the bootstrap) and '`vrbs-1.1`' (the loader). The bootstap is loaded to EEPROM and stays resident until another bootstrap is loaded. The loader is temporary and is only needed because you cannot simultaneously write to and run from EEPROM.

Run arc at 38400 baud. For example, `arc -port <port> -baud 38400`. Definitely do not use the `-saphira` or `-saphira_stop` options as these will cause the baud rate to be set at 9600, which will not work for this application.

```
arc> cd /usr/local/arc/332libs-0.986
arc> bsload vbs-1.1 vrbs-1.1
This command loads a bootstrap into your EEPROM If your
EEPROM already has a bootstrap, you shouldn't need to do this.

If you a kernel version 0.980 or earlier, you are not currently using
a bootstrap in your EEPROM, and will need to install a bootstrap
in order to upgrade your kernel.
```

```
Warning:  Don't turn your board off during this download, and
do not attempt this download over a radio modem.  If the download fails
your EEPROMs may no longer boot, and you'll need to get replacements

Do you wish to download a boostrap at this time? (y/n)
```

Answer yes. It will then first download the loader to RAM:

```
Download vrbs-1.1 (a.out file, gdb protocol)
NOT LOADING SYMBOL TABLE.
Downloading code (0x107000-0x107f5f)..............
Downloading global initializers (0x107f60-0x107f7f).
arc>
```

Then it will run the loader and use it to load the bootstrap:

```
Running loader (at address 0x107000)

EEPROM LOADER 1.1
Calling load command
in download_aout_rest
Download vbs-1.1 (a.out file, gdb protocol)
NOT LOADING SYMBOL TABLE.
Downloading code (0x0-0xec3)Writing 0-3F...
.Writing 40-7F...
.Writing 80-BF...
```

When it is done, it will print:

```
Bootstrap download completed.

Please reset or power-cycle board.
```

Now you will have a new bootstrap installed. If you were installing a new bootstrap with a post 0.981 kernel, the kernel will probably start up when you reset. If you were installing the first bootstrap in EEPROMs with a pre 0.981 kernel, you will need to use `kernelload` to download the kernel first. Until you do this, it will print the boot message ever second or so while on. This is normal, and means that it is ready to receive a new kernel. See Section 3.4.3 [Loading a new kernel using ARC], page 25 for details.

4 Compiling programs under ARC

This section assumes that you have already gone through the section on getting started. See Chapter 2 [Getting started using ARC], page 7 if this is not the case. This section provides more detail on topics involved in compiling programs under ARC.

4.1 Specifying which libraries and kernel to use

It is important to specify which kernel to compile against before trying to compile a program under ARC. You can make this specification at several different levels:

- You can give a command-line option to `arcc`, either directly or by putting it on a line in a '`.lis`' file.
- You can set up your own default in your '`~/.arcrc`' file.
- You can use the default kernel specified by a symbolically link called '`default`' in a subdirectory of the main '`arc`' directory.

The options in this list are looked for and used in order. For example, if you specify a command-line option to `arcc`, the other options are not pursued.

4.1.1 Choosing the kernel which is right for you

Kernels are stored in subdirectories of the main '`arc`' directory and have the following form:

'`arc/332libs-x.xxx/motoboot.xxx`'
: A Motorola 68332 kernel which will run on a Motorola BCC332 board

'`arc/332libs-x.xxx/vestaboot.xxx`'
: A Motorola 68332 kernel which will run on a Vesta SBC332 board

'`arc/332libs-x.xxx/ttaleboot.xxx`'
: A Motorola 68332 kernel which will run on an Onset Tattletale 68332 board

'`arc/386libs-x.xxx/pckernel.xxx`'
: An Intel 386, 468, Pentium kernel which will run on an IBM PC (not yet in distribution).

To discover which version a given board is running, look at the tag on the ROMs, or connect the board to `arc`, hit reset, and look at the boot line. It should have the version number in the boot message.

```
Anarchboot Multitasking 68332 Kernel Version 0.919
Copyright Anne Wright 1993

Compiled May  3 1995 at 13:09:12

vestaboot.919>
```

This example is from a Vesta SBC332 board. It is running version 0.919 of the 68332 kernel for Vesta boards, so if you were using this board, the kernel you want is '`vestaboot.919`'.

4.1.2 Specifying which kernel in a project-specific way

If you are likely to want to switch between different kernels frequently, or if certain projects use a kernel other than the default, you should include `KERNEL=<kernelname>` as an argument to `arcc`, or on a line in the `.lis` file for the project.

An example of using this in a Makefile is:

```
foo: foo.c
        arcc KERNEL=vestaboot.919 foo.c

bar: bar.c
        arcc KERNEL=ttaleboot.915 bar.c
```

In this case, when you execute `make foo`, it will use '`vestaboot.919`' as the kernel, but when you execute `make bar`, it will use '`ttaleboot.919`'.

You also could have used '`.lis`' files:

foo.lis:

```
KERNEL=vestaboot.919
foo.c
```

bar.lis:

```
KERNEL=ttaleboot.915
bar.c
```

In this case, when you execute `arcc foo.lis` (either by typing the command at a shell or loading it from `arc`), it will use '`vestaboot.919`' as the kernel, but when you execute `arcc bar.lis`, it will use '`ttaleboot.919`'.

4.1.3 Specifying your personal default kernel

If you are not likely to want to switch between different kernels frequently, or if you will be using one kernel predominantly, you can specify your default kernel in '`~/.arcrc`', the ARC resources file in your home directory.

Just create '`~/.arcrc`' and include the line `KERNEL=<kernelname>` on a line by itself. An example '`~/.arcrc`' file is as follows:

```
KERNEL=vestaboot.919
```

If you specify which kernel to use as a command line option or in a '`.lis`' file, this default will be overridden.

If you neither specify the kernel at the command line or in a '`~/.arcrc`' file, then the global default kernel (if any) will be used. See Section 1.6 [Set up the default kernel], page 5 for details.

4.2 Using arcc

The actual compiler being used to compile code under the ARC Development system is `gcc`, the GNU C Compiler. `gcc` is free, copy-lefted software. The source is freely available via anonymous `ftp` from '`prep.ai.mit.edu`' in '`pub/gnu`'. We distribute `gcc` cross-compilers with our system as a convenience, but we cannot and do not charge anything for it.

However, in order to get `gcc` to compile and link correctly for this specialized use requires a large number of command-line options and paths. This used to be done via complicated Makefiles and shell scripts.

To make the process of compilation easier, we wrote a front end called `arcc` which knows how to find the correct directories for libraries and header files, and set up the options for `gcc` correctly.

(This does, however, require you to set up your environment properly. See Section 2.1 [Set up your environment to use ARC], page 7 for details.)

For example, if you have a file called 'hello.c' and you have set up the default kernel already, you can type `arcc hello.c` and it will create an executable 'hello'. Calling the `gcc` cross-compiler directly, you would have to type, for example:

```
332gcc -nostdinc -nostdlib -B/usr/local/arc/sparcbin/332
-I/usr/local/arc/332libs-0.9 -m68020 -mnobitfield
-freg-struct-return -g -O -L/usr/local/arc/332libs-0.9
-Wl,-R,/usr/local/arc/332libs-0.9/vestaboot.919,-N
-Wl,-Ttext,106000 -Wl,-T,/usr/local/arc/332libs-0.9/usercode.ld
/usr/local/arc/332libs-0.9/usercrt0_332.o hello.c -o hello -lgcc
-ldefaults -ltpu
```

arcc options

The following options can be specified at the `arcc` command line, placed on a line by themselves in a program's '.lis' file, specified in a person's '~/.arcrc' file, specified in the '.info' file for libraries or kernel, or (in the case of KERNEL) in the filesystem of '/usr/local/arc'. The above are listed in the order of precedence. Options specified in more than one place will take the value from the command line if specified, otherwise from the '.lis' file if specified, etc.

'KERNEL=<kernelname>'
: Link against the specified kernel, and use it to determine which directory to find header files and libraries in. For more information, see Section 4.1 [Specifying which libraries and kernel to use], page 29.

'CODE_ADDR=<address>'
: Set the starting address of the program to <address>. If not specified at the command-line or in your '~/.arcrc' file, the default is to use the value of 'CODE_ADDR' specified in the '<kernelname>.info' file in the same directory as the kernel itself.

  The default radix for the address is decimal, so be sure to specify '0x' at the beginning for hexidecimal addresses. Use the `memmap` console command to determine where free space is on the board (see Section 5.4 [User console], page 40).

'LSB_FIRST'
: Specify this if you are targeting an LSB first processor, for example, the Intel x86 family. **Do not use this option for using the Motorola 68332.**

'SHOW_ARGS'
: Show the full arguments to gcc (cannot be used as a command-line option).

4.3 Changing program startup parameters

Your program can optionally specify certain parameters that affect how it is run.

'`run_on_reset`'
: This detemines whether you want the function "main" in your program to be run when the reset button is pressed. A value of 0 for run_on_reset specifies that your program should not be run on reset, and a non-zero value specifies that it should be run on reset. The default value is 1.

'`startup_stack`'
: This determines how much RAM will be allocated for the stack in your program's main process. The default is 2048, but if you get "stack overflow" messages from the board, you should specify a higher number.

'`startup_ticks`'
: This determines how many ticks your program will take in the multitasking system. Usually, you won't need to change this. The default is 5. By specifying a higher number, your program's main process gets to run for a longer time before another process gets a chance.

You can override the defaults for these values by defining a global variable with the new value.

Examples:

```
int run_on_reset= 0;     /* don't run main() on reset */

int startup_stack= 4096;  /* give me 4K of startup stack */
```

Like any other global variable, you should only declare these parameters once in each program or you will get compile errors.

5 Kernel

Eventually, this chapter will describe the ARC kernel in detail. However, at this time it is still under construction, and is sketchy in many places. Over time it should improve.

5.1 Kernel Overview

The ARC kernel is the part of the ARC development system which runs on the embedded processor. The kernel boots the system and provides a user-interaction console, a remote downloading and debugging interface, a multitasker, and libraries which can be shared by user programs. The full system currently runs on Motorola 68332 processors. A partial port (not including a multitasker or boot code) also runs on Intel 386, 486, and Pentium processors.

In creating the ARC kernel, I sought to provide powerful debugging support and multitasking control with a minimum of system overhead in order to facilitate real time appliations. I also sought to allow multiple processes to use shared resources, like the serial port, in a clear and efficient manner.

The kernel performs system services and facilitates the loading and running of programs, which are key features of any operating system. However, unlike most other operating system it does not put up any firewalls that isolate user programs from access to the hardware or the kernel internals. When a user program is compiled, it is linked to an image of the kernel. The user program therefore has direct access to the kernel globals and functions.

This system has the advantages that it is more efficient at runtime and allows more complete code sharing and lower memory usage than the standard way operating systems are implemented. However, it also has some disadvantages. It is potentially easier for a user program to mess up the operating system's state by allowing it complete access to the operating system, and it makes the problems of version skew worse.

The standard way for an operating system to provide services to client programs is for there to be some library code linked into the user program which sets up and performs a trap (a type of interrupt). The trap allows the operating system to take control of the processor in a well defined manner and provides a way for the user code to interact with the operating system without having access to its internals.

However, the overhead for performing a trap is much greater than the overhead for a normal function call. I believe that the added security of using traps is not worth the extra time expense when the goal is to make good real-time systems, as was my goal in designing the ARC kernel.

Also, because the ARC kernel is multitasking and the user interaction and debuggging processes are independent of any user processes, even if user processes crash, the supervisory processes are still there to help you debug your crashed process. Without having hardware memory protection, you could not really do better than that (unless you store the interrupt table in ROM) even if the only interaction between the user processes and the operating system were through traps. Without hardware memory protection, there is no way to stop a user from overwriting the interrupt vector table, which could bring the operating system in either method to its knees.

On reset, kernel sets up necessary system resources. It initializes the chip selects, globals, and serial port; checks the checksums on the memory map and persistant table; sets up multitasking, the TPU, the memory manager, the console process, and the gdb process; then it checks the memory map for programs which wish to be run on reset. After this, the console, gdb, and user programs are free to be used.

5.2 Memory Management

Because ARC is intended to primarily run on embedded controllers, "memory management" refers only to the way the kernel manages the contents of the RAM. Blocks cannot be swapped out to disk because there is no disk. Because of this, the memory management facilities are fine-grained. They are also geared towards allowing programs to persist across reset and power cycling so long as the RAM is not unduly corrupted.

The memory management consists of two modules – the memory map, which is relatively coarse-grained persists across resets, and malloc, which is fine-grained and does not persist across resets.

5.2.1 Memory Map Module

The memory map keeps track of the name and type of data contained in the memory. It can manage multiple devices, and keep track of whether each device is RAM or ROM. It is not currently possible for users to add their own devices to the memory map, but I'm working on it. However, the current kernel does automatically detect the size of the RAM and ROM in the system and bounds the memory map appropriately.

When the board is reset, the kernel checks whether whether the memory map is still valid. It determines this by comparing the stored checksum against a freshly computed one. If the stored memory map is valid, it uses that. If it is not valid, the kernel creates a new memory map with default sizing for the system tables and heap. It also searches through ROM and RAM to find programs which are still there and valid, and adds those to the map.

Because the kernel and the user programs will be potentially sharing a small amount of memory, blocks may be specified to start at any address and may be of any size. Use the `memmap` console command to view the memory map.

A typical memory map looks like this:

```
Start   End     Size    Type         Name          Device
000000  012393  012394  Code         vestaboot.919 ROM
012394  0125c3  000230  Data         Kernel inits  ROM
100000  1003ff  000400  Vector Table Vector Table  RAM
100400  100dff  000a00  Memory Map   Memory Map    RAM
100e00  102603  001804  Data         Persistents   RAM
102f00  105a9f  002ba0  Data         Kernel data   RAM
105aa0  105e9f  000400  Data         GDB Block     RAM
106000  1062ef  0002f0  User Code    mobot-vision  RAM
1310a8  13f7ff  00e758  Heap         Heap          RAM
13f800  13ffff  000800  Stacks       Root stack    RAM
```

Blocks which are not being used for programs, data, or system tables are used by malloc. Therefore, if you want to use a block of data, you should first reserve it by using the `memblock` console command (see Section 5.4.3 [Viewing memory usage], page 42).

5.2.2 Malloc Module

The malloc module manages blocks from the memory map, and partitions them up when `malloc()`, `realloc()`, or `free()` are called.

The "Heap" block in the memory map is always reserved for use by the malloc module and you are prevented from downloading a program in the block reserved for it. However, to achieve maximum usage of a potentially small amount of memory, the malloc module incorporates unused blocks in the memory map for temporary use. Temporary use in this case refers to memory which does not persist across downloads.

The output of the '`malloc`' console command (done during the same session as the memory map in the previous section) shows the temporary heaps:

```
Heap 0x1062f0, size 175544 (0x2adb8)
  0x1062f8:     Free  real size=175536
Total free = 175536, Total used = 0
Heap 0x102604, size 2300 (0x8fc)
  0x10260c:     Free  real size= 2292
Total free = 2292, Total used = 0
Heap 0x1310a8, size 59224 (0xe758)
  0x1310b0:  Alloced  real size=   38   PID=0
  0x1310d6:  Alloced  real size=  270   PID=0
  0x1311e4:  Alloced  real size=   38   PID=0
  0x13120a:  Alloced  real size=  270   PID=0
  0x131318:  Alloced  real size=   42   PID=0
  0x131342:  Alloced  real size=   28   PID=0
  0x13135e:  Alloced  real size= 2062   PID=0
  0x131b6c:  Alloced  real size=   38   PID=0
  0x131b92:  Alloced  real size=  270   PID=0
  0x131ca0:  Alloced  real size=   42   PID=0
  0x131cca:  Alloced  real size=   30   PID=0
  0x131ce8:  Alloced  real size= 1038   PID=0
  0x1320f6:  Alloced  real size=   38   PID=33
  0x13211c:  Alloced  real size=   54   PID=33
  0x132152:  Alloced  real size=   98   PID=33
  0x1321b4:     Free  real size=   98
  0x132216:     Free  real size=54762
Total free = 54860, Total used = 4356
-----------------------------------------------------
Total free = 232688, Total used = 4356
```

The first two heaps are temporary and were allocated in holes in the memory map. The last heap, and the only one which has been used in this example, is the "Heap" block. In this example, the board has a total of 256K of memory, of which less than 30K is being used by system overhead and a small user program. Malloc is managing 230K of free memory, most of which has been recovered from blanks in the memory map.

These optimizations have the following repercussions:

- Memory malloced by the kernel itself is always allocated from the main Heap because it needs to persist across downloads.
- By default, memory allocated by user processes is deallocated when the process which malloced it dies (or is killed). Because processes by default die when their parents die, it is safe for child processes to use memory allocated by a parent. However, it is not in general safe for processes

to use data malloced by other proceses without first checking that the process which malloced the memory has not died.

- By default, memory allocated by user processes is allocated from the temporary heaps.
- To provide for situations in which memory allocated by user processes needs to live longer than its creator process, the `durable_malloc` function was created (see Section 6.4 [Malloc functions], page 51). The pointers it returns are in the main Heap and are not automatically freed (except on reset). Therefore, you should be careful when using durable_malloc to avoid creating memory leaks.
- User programs cannot use empty blocks in the memory map as static, non-malloced memory (you used to be able to get away with this in old versions, though it was not recommended). You should first reserve memory blocks by using the `memblock` console command (see Section 5.4.3 [Viewing memory usage], page 42).

5.3 Polling vs. Interrupts

Responding to asynchronous events is one of the most basic and necessary functions of a computer system. There are two fundamental approaches to dealing with asynchronous events: polling and interrupts.

In a polling scheme, you typically have a flag which is set by an asynchronous event and polled by the processor. The processor periodically checks the status of the flag. When the processor sees that the flag is set, it takes the appropriate actions (reads the data from a latch, writes data to a latch, turns on a motor, etc.) and clears the status flag. Polling works, but can take a lot of processor time if you want to respond to an event quickly.

Interrupts take advantage of special hardware in a processor which allows normal processing to be preempted by asynchronous events. Using interrupts to process asynchronous events has the advantage that the processor only spends time servicing the event when it actually happens. Therefore, for the same desired response time, the CPU overhead for interrupt servicing can be much lower.

The disadvantages of using interrupts are that there is more setup involved and it is typically harder to debug interrupt routines. Therefore, it is often a good idea to start out with a polling scheme and, when you feel confident that the servicing routines work, install them on interrupts.

Typically there will be a range of possible interrupt sources, each of which is assigned a number. An interrupt vector table, located in memory, stores the address of a handler for each possible interrupt. When an interrupt occurs, the processor saves some of the current processing state (which typically includes the status flags, program counter, and interrupt source, and may include other registers or data), looks up the address of the correct interrupt handler, and resumes execution in the handler.

The handler must save any registers it will use before it starts. The handler may also need to perform some action to clear the interrupt request (this is usually the case with hardware interrupts). When the handler is done, it restores the registers to their original condition and performs a return-from-interrupt instruction to return to the processing that was preempted.

5.4 User console

The console is the default interaction mode of the root process. The model for using the console is that of a unix shell. At the console you can view and modify the contents of memory, run programs, view and kill processes, view and modify the value of persistents, view the memory map, list and unload programs, change the baud and clock rates, and much more. Commands are typed at the console prompt and those commands are executed by the root process. If any console command hangs, pressing the abort button (an active low button attached to the IRQ7 pin) will return you to the interaction prompt.

If not otherwise specified, assume that all numerical arguments are in hexidecimal.

5.4.1 Command summary

'`help`' Print help information

'`run`' Run program: run <name> [<stack>] [<ticks>]

'`go`' Run code at addr: go <addr> [<stack>] [<ticks>]

'`resetrun`'
Run reset programs: resetrun

'`memmap`' Display memory map

'`devmap`' Display device map

'`memfree`' Display free memory

'`memblock`'
Create memory block: memblock <name> <addr> <len>

'`memrm`' Remove memory block: memrm <name>

'clearmem'
: Reinitialize memory map

'checksum'
: Checksum: <name> or <start> <size>

'list'
: List programs currently loaded

'ls'
: List programs currently loaded

'unload'
: Unload program: unload <name>

'remove'
: Remove program from memmap: remove <name>

'restore'
: Restore program: restore <name>

'malloc'
: Print memory allocation history

'ps'
: Display process table

'kill'
: Kill process: kill <ps>

'fix'
: Fix process: fix <ps>

'files'
: Display active files table

'streams'
: Display active streams table

'md'
: Memory Display: md [<addr>] [<length>]

'mm'
: Memory Modify: mm [<addr>]

'rd'
: Display registers: rd [<ps>]

'set'
: Set values: set <var> [<val>], default val = 1

'show'
: Show values: show <var>

'persist'
: Show/set values of persistents

'time'
: Display value of time counter

'quit'
: Quit

'baud'
: Set baud rate: baud <baudrate>

'clock'
: Set clock rate: clock <hertz>

'tickperiod'
: Set tick period: tickperiod <ns>

'tpu_interrupts'
: Enable/disable tpu_interrupts

5.4.2 Running programs

'run <name>'
: Run program by name. The program is checksummed and its dependencies are checked before running. If the checksum fails, you are not allowed to run the program. If the dependencies fail, you are prompted whether to run it or not. The program is run as a new process. The amount of stack and number of ticks are taken from the startup_stack and startup_ticks variables in the user program. (See (see Section 4.3 [Changing program startup parameters], page 33) for details).

5.4.3 Viewing memory usage

'memmap'
: Display memory map. This is your only real way to know how the memory of your 332 is being used. Unfortunately, you have to manually know where the free blocks in your memory are and set up the -start option to the link script accordingly. Therefore, knowing the memory map on your board is critical, and this is the command that gives it to you.

'memfree'
: Display free memory blocks. Free memory blocks are used as heap between downloads, but this will show you where you can download programs to or reserve data blocks with the memblock command.

'memblock <name> <addr> <len>'
: Create a reserved data block in the memmap with name, starting address, and length as specified. The address and length are interpreted as being in hex. This command causes all processes to be killed and the process heaps to be reallocated to clear the reserved memory block from being changed by malloc.

'memrm <name>'
: Remove named memory block. If there is more than one block with the same name, the first one will be removed. This can be used for executables, but is really intended for data blocks since executables can be deleted with unload or remove.

'checksum <name>'

'checksum <start> <size>'
: If given a name arguments, checksums the named block. Otherwise checksums starting at the start address for the size given. The checksum is just a simple add-every-byte-together checksum. The result is just printed.

'list'
: List programs currently loaded. The checksum/version number is also shown for convenience in checking dependencies.

‘unload <name>’
: Unload program and delete it from the memory map. If there is more than one program in memory with the same name (a situation which should really be avoided), it will unload the first one in memory.

‘remove <name>’
: Remove a program from the memory map, but do not destroy it. So long as the memory block is not overwritten, subsequent calls to clearmem will restore it.

‘restore <name>’
: Attempts to restore the executable of the name specified. If there is backup in the memory map named <name.bak>, that program will be copied to the location where it is supposed to be.

  This is mainly useful for when you want to recover a version of a program that is in ROM. Programs can be stored in ROM, but the can currently only be run out of RAM. In fact, they can only be run if they are in the place in the ram where they were linked to be. The reason that code is not relocatable once it is downloaded to the board is that only the actual executable and globals are downloaded to the board. The symbol table, etc. which is used to relocate code only on the host and most likely would not fit on the board anyway.

‘malloc’
: Print memory allocation history. The pointer, size, and creating process are printed for each currently mallocced entry. This could be useful for debugging.

5.4.4 Process management

‘ps’
: Display process table. Use this a lot. It's extremely useful. The pid, status, stack usage, current pc, and name are all printed.

‘kill <ps>|<name>’
: Kill process. The process can be specified either by its pid or its name.

5.4.5 Viewing and modifying memory

‘md [<addr>] [<length>]’
: Memory display. Address and length are specified in hex and are optional. If no address is specified, the addresses following the last ones displayed by the md command are used. If no length is specified, 16 bytes are displayed. It is not possible to specify

a length and not an address, as the first argument is assumed to be the length. Immediately after an md command is issued, subsequent blank returns are interpreted as md.

'`mm [<addr>]`'
: Memory Modify. If no address is specified, the address after the last one modified is used. Memory is modified on a word by word basis, and the location is read back to confirm that the data stored properly. Entering a blank line is assumed to mean that that the specified word should not be altered. Entering a '.' on a line by itself exits out of mm mode.

'`rd [<ps>]`'
: Register display. Displays the current registers for the specified process id. If no process is specified, the root process' registers are displayed.

'`persist [<peristname>] [<value>]`'
: Show/set values of persistents. If no arguments are given, all persistents are printed. If just a name is given, that persistent only is printed. If a value is given, the value of the persistent is set to that value. If the persistent does not exist, you are given the option to create it.

'`set <varname> [<val>]`'
: Set debugging values or persistents.

'`show [<varname>]`'
: Show values. This is for kernel debugging flags. If no arguments are specified, all possible values are printed. Otherwise, just the requested one is printed.

'`time`'
: Display time. The time is persistent and increments from the moment your persistent table was last initialized. This is how you look at it. Your programs can access the time by getting its pointer by a call to lookup_persistent("time") and dereferencing the pointer.

5.4.6 Setting system parameters

'`baud <baudrate>`'
: Set serial baud rate. The baudrate argument is in decimal. Using multiterm's baud command causes multiterm to call this command on the board and change its own baud rate simultaneously. Otherwise, you will have to call this command, then manually change host baud rate before being able to talk to the board. The baud rate is persistent, so it will reset to the same value it was before on reset or power up. If you set the board to a bad or unknown baud rate, there is no really good way to recover aside from removing power from your rams, which will cause all your programs and

persistents to be wiped. If anybody has a good suggestion for an override on this, please tell me.

'`clock <hertz>`'
: Set clock rate. The frequency argument is in units of hertz and read as a decimal number. The 332 will by default be run at 16777216 hertz. However, the clock rate is controlled by a programmable VCO which can produce a wide range of clock frequencies down into the kHz range. Running the clock faster than 16777216 hertz can be done, but this is the specified maximum and bad things can happen at faster speeds. The baud rate is automatically adjusted to compensate for the clock rate, but be aware that as the clock rate decreases, the maximum supportable baud rate decreases. If you change the clock and lose serial, reset and try again at a lower baud rate. Also be aware that the scheduler tick period should be increased when the clock frequency is decreased.

'`tickperiod <ns>`'
: Set scheduler tick period. The period argument is in nanoseconds and read as a decimal number. As a benchmark, as of March, 1994 the scheduling interrupt during a swap takes 120us at a clock rate of 16MHz, and 6us when a process has not yet run out of ticks. Therefore, with a 1ms tick period, giving only 1 tick per process, and using no defers 12% of the CPU is taken up by context switching. This percentage decreases as the tick period increases, at the expense of scheduling granularity. This percentage decreases as the average number of ticks per process increases because fewer of the scheduling interrupts result in a swap. This percentage increases as the clock rate decreases, because the same amount of computation takes longer. Therefore, I would suggest a good range for the tick period as 1000000ns (1ms) which is the default and up. The system "time" is in units of shecduler ticks, so changing this parameter also affects timing granularity.

5.4.7 Miscellaneous

'`help`'
: Print help information.

'`quit`'
: Quits the console. This is only really useful if you are running a ram kernel and you want to escape back to rom kernel.

6 ARC Programming Reference

6.1 Standard input/output functions

These functions read or write from the standard input and output streams for a process. By default, the standard io streams will be multiplexed serial streams.

Standard ANSI stdio functions

'`int getchar();`'

'`int getc(FILE *stream);`'

Both of these functions return the next character from the input buffer that has not yet been read. If there is an error, like when the stream argument is invalid, a -1 is returned. getchar() returns the next character from the default input stream for the process calling it. getc() returns the next character from the given stream. getchar() is frequently used in programs to wait until the user is ready to continue (The old "press any key to continue" prompt). Keep in mind that a buffering getchar will cause execution to continue if there is any past input left in the buffer, so you may want to call clear_serial_output_buffer() immediately before the call to getchar for this purpose.

'`int putchar(char ch);`'

'`int putc(char ch, FILE *stream);`'

Both of these functions write the given character to the output queue to be sent over the serial line. Because serial output is buffered, these functions will return after writing to the output buffer. If the buffer is full, these functions will not return until there is sufficient space in the buffer to write the next character. Because all other serial output functions (puts(), printf, and its derivatives) call putchar, this applies to all of them.

'`char *gets(char *s);`'

Gets a string from over the serial line, echoes it, and places it into the character array s. s must have enough space allocated to handle the input or overflow will occur. gets() takes care of delete and backspace characters appropriately and returns the pointer to the null terminated string upon receiving a return character. I recommend that you do not use gets(s), but rather use fgets(stdin,length,s) with length set to be the space allocated to s. This is a much safer in situations where overflow is possible.

'`int sscanf(char *s, char *format, ...);`'
: Standard sscanf which scans the input string s and writes the appropriate values to it's arguments. A modification of this beyond standard is that a %b format string parses the binary representation of a number.

'`int scanf(char *format, ...);`'
: Standard scanf which calls sscanf with the results of a gets. The input length is limited to 100 characters.

'`int puts(char *ptr);`'
: Writes a null terminated character string to standard output by calling putchar on each character.

'`int printf(char *format, ...);`'
: Standard printf (look in a C reference for specs) using putchar to output each character to standard output the modification that %#b prints the binary representation of a number to # (in the range 1 to 32) binary digits. If # is left out, the defaults is to display 8 bits.

'`int sprintf(char *s, char *format, ...);`'
: Standard sprintf which writes to the character string s instead of an output stream.

Kernel extensions of stdio facilities

'`void clear_serial_input_buffer();`'

'`void clear_serial_output_buffer();`'
: These functions clear the I/O buffers. This will result in data lossage, but sometimes you want that.

'`void wait_for_serial_output_complete();`'
: This function returns when the standard output buffer is empty.

'`int serial_data_received();`'
: Returns a 1 if there is data in the stadard input buffer, otherwise returns 0. This is a useful loop condition if you want a keyboard hit to cause a loop to exit.

'`int serial_input_buffer_depth();`'

'`int serial_output_buffer_depth();`'
: These functions return the depth of the standard I/O buffers. The input buffer depth is the number of characters received but not yet recovered through a call to getchar(). The output buffer depth is the number of characters written to the output buffer but not yet removed from the buffer. If the standard output is a multiplexed stream, this depth may reflect the activity of other processes as well.

6.2 File input/output functions

These functions open, close, read from or write to files and streams.

The file abstraction module allows io to be treated in similar way to how it would be treated on other c systems. There are still no real files (the memory is not permanent or large enough to really justify it), but the file abstraction provides a consistent interface to io drivers.

The "file" name structure simulates a 2-deep directory hierarchy where the "directory" name is the driver and the "filename" is the name of the stream. Once created with fopen, files exist until all instances of it are destroyed or until reset. If a file with the same name is opened in multiple progrograms, processes, etc. all will get a pointer to the same file object.

This allows for inter-process communication streams, or multiple processes watching the same hardware io stream, etc. However, once anyone removes a character from the file it's gone. There are not independent file pointers.

Files have an input and an output queue (which may point to the same place) which are fixed-length circular buffers. If a write is attempted to a queue which is full, the write will either hang or return -1 depending on the file type.

There are currently four drivers: stream, serial, seriala, and serialb.

- Stream is a wrapper around a single queue. Writes to the stream insert characters into the queue, reads read characters from the queue. Usage: '`fopen("/stream/foo","");`'
- Serial files do the obvious thing: writes send data out the serial port and reads read from the serial port. It is possible to specify "R" in the mode string to fopen (arg 2) in order to get a raw stream. A raw stream gets every character that is received at the serial port and sends characters directly to the serial port. If "R" is not specified, the multi-threaded stream protocol (used by arc and multiterm) is applied to the stream. Usage: '`fopen("/serial/foo","")`' or '`fopen("/serial/foo","R")`'.
  - '`/serial/foo`' gets the default port specified by the default_port persistent (0 for a, 1 for b).
  - '`/seriala/foo`' always gets porta.
  - '`/serialb/foo`' always gets portb (if it exists).

The standard files stdin, and stdout refer to the serial stream on the default port which is created automatically for the current process. The standard file stderr refers to the console stream.

The following standard functions are available and should do what you expect:

- FILE *fopen(const char *filename,const char *mode);
- int fclose(FILE *s);
- int fgetc(FILE *f);
- int fputc(int c,FILE *f);
- int fpeekc(FILE *f);
- int fgets(FILE *f,int n,char *s);
- size fread(char *buff,size_t element_size,size_t count,FILE *f);
- size fwrite(char *buff,size_t element_size,size_t count,FILE *f);
- int fputs(FILE *f,char *s);
- int fprintf (FILE *f, char *format, ...);
- int vfprintf (FILE *f, char *format, va_list arg);
- int vsprintf (char *s, char *format, va_list arg);
- int vsputc (char c, FILE *f);
- int vsscanf (char *s, char *format, va_list arg);
- char *gets(char *s);

The following I defined because I thought they would be useful:

'`int fflush(FILE *s);`'
: Waits until the output buffer is empty.

'`int fclear(FILE *f)`'
: Clears the input queue to file f.

'`int fdepth(FILE *f)`'
: Returns the number of bytes in the input queue of the file.

6.3 String/memory functions

The following functions are defined as specified in "C A Reference" to conform to ANSI standard:

- String functions
  - char *strcat(char *dest,const char *src);
  - char *strncat(char *dest,const char *src,size_t n);

  - int strcmp(const char *s1,const char *s2);
  - int stricmp(const char *s1,const char *s2);
  - int strncmp(const char *s1,const char *s2,size_t n);
  - char *strcpy(char *dest,const char *src);
  - char *strncpy(char *dest,const char *src,size_t n);
  - size_t strlen(const char *s);
  - const char *strchr(const char *s,int c);
  - const char *strrchr(const char *s,int c);
- Memory functions
  - void *memchr(void *ptr,int val,size_t len);
  - int memcmp(void *ptr1,void *ptr2,size_t len);
  - char *memcpy(void *dest,void *src,size_t len);
  - void *memset(void *ptr,int val,size_t len);
  - void memswap(void *a, void *b, size_t len);
- Sorting functions

```
void bssort(void *base, size_t n_elems, size_t elem_size,
            int (*compare)(void *, void *));
```

  This has the same arguments as standard qsort, but does a bubble sort instead. There is no standard qsort at this time.

6.4 Malloc functions

'`void *malloc(size_t size);`'

'`void free(void *mem);`'

These functions are as documented in C texts. malloc() takes a integral size as an argument and returns a pointer to a block of memory of that size, or NULL if there is not enough memory left. free() takes a pointer returned from a previous call to malloc() and frees that block of memory for reuse. Call free with a non-malloced pointer or a previously freed pointer at your own risk. If free detects that the pointer is not valid it will breakpoint. You may then want to use gdb to find out where it crashed.

Note that all of the memory malloced by a process is automatically freed when that process dies. Therefore be careful not to use memory malloced by one process in another process unless you are confident that the proces which did the malloc has a longer lifetime. It is in general safe for child processes to use memory which the parent had mallocced because they will die with the parent.

'`void *durable_malloc(size_t size);`'

This function is just like malloc except that memory allocated using durable_malloc is not automatically freed when the process which allocated it dies and may exist even across downloads. It is possible for durable_malloc to fail when regular malloc would succeed, as it has a smaller heap to allocate from, so you should use malloc when you don't need this extra feature.

6.5 Persistents

Persistents provide a way to store state between resets and downloads, implement inter-program communication, and adjust program parameters from the console.

A persistent is really a permanently allocated 4-byte memory location associated with a unique string. Persistents are created by calls to '`init_persistent()`' and stored in a hash table. Any program may call '`init_persistent`' or '`lookup_persistent`' and receive a pointer to the same permanently allocated memory.

Persistents age at every reset, and persistents which have not been accessed in long enough (default of 10 resets) are garbage collected.

The type '`persistent`' is a pointer to a volatile integer. The following is a sample usage of persistents:

```
persistent foo;
void main()
{
  foo=init_persistent("foo",4);
  if(foo)
    printf("Value stored in foo is %d\n",*foo);
  else
    printf("Persistent table full\n");
}
```

The size of the persistent table is reconfigurable, but requires a recompile of kernel. The default is 256 entries with 16-character hashing strings, which comes to 8K of memory usage.

'`persistent init_persistent(char *s,int defval);`'

Look up the string s in the persistent table. If it is already there, return the associated memory pointer. If it is not already there, create it with the provided default value.

If there are no slots left in the persistent table, returns NULL. If the string is too long, only the first N characters are used, where N is a length built into the kernel – defaulting to 16.

'`persistent set_persistent(char *s,int val);`'
: The same as init_persistent, only the value is set to val regardless of whether it was already there or not.

'`persistent lookup_persistent(char *s);`'
: Looks up the string s in the persistent table and returns either the associated memory pointer or NULL if it was not found.

'`int delete_persistent(char *s);`'
: Deletes the persistent from the persistent table on the next reset by setting its age to be ancient. It does not delete the entry due to the danger that someone else might have the handle to the persistent.

'`void print_persistents();`'
: Prints the entire persistent table. This function is called by the "persist" console command with no arguments.

6.6 Queues

A queue is a first-in-first-out (FIFO) data structure which is specially designed for use in the multitasking environment with shared memory which kernel provides. So long as only the accessor functions '`read_byte_from_queue()`' and '`write_byte_to_queue()`' are used, queue reads and writes are atomic. Therefore queues can be used by multiple processes without extra locking. Queues use interrupt masking to achive atomicity, so it is safe to use them in interrupt handlers (unlike '`lock()`')

'`void initialize_queue(queue *qp,int size,unsigned char *data);`'
: Given a queue pointer, a size, and a pointer to a block of data of the specified number of bytes, initialize_queue() initializes the queue to be empty. Because queues must conform to certain representation invariants in order to work properly, they should be initialized before they can be used.

'`int init_queue(queue *qp,int size);`'

'`int free_queue(queue *qp);`'
: init_queue() is just like initialize_queue() except that it mallocs the data buffer of the specified size. If there is not enough memory left to allocate the data, returns -1, otherwise returns 0. There are times you want to use this for convenience. A queue

that has been initialized using init_queue must be freed using free_queue() in order to free its storage (or you could just free qp->data, but this has better abstraction qualities).

'queue *create_queue(int size);'

'int destroy_queue(queue *qp);'
: These two functions are the same as the ones above, but they will allocate and free the queue as well. It is not really necessary to provide all of these functions, but it's basically free to do so and there are occasions that require each of these methods.

'int write_byte_to_queue(queue *qp,unsigned char b);'
: Write a byte to the queue. If the queue is full, then the status code `QUEUE_FULL` will be returned. Otherwise, b will be atomically added to the end of the queue.

'int read_byte_from_queue(queue *qp);'
: Read a byte from the queue. If the queue is empty, then the status code `QUEUE_EMPTY` will be returned. Otherwise, it will return the first byte in the queue. (The type is int instead of char so that error conditions are unambiguous.)

'int peek_queue(queue *qp);'
: Returns `QUEUE_EMPTY` if the queue is empty, otherwise returns the next byte in the queue without removing it. You are not guaranteed that the same byte will be there on a subsequent read if another process is trying to read the queue.

'int queue_depth(queue *qp);'
: Returns the number of bytes currently in the queue.

'void clear_queue(queue *qp);'
: Throw away the current contents of a queue and make it empty.

6.7 Locks

A concern in multitasking systems with shared memory is achieving mutual exclusion of certain operations. For example, if you had a insertion/deletion table, you would want only one process to be able to add an entry at a time to prevent the possibility of two processes trying to add different entries to the same slot at the same time. The functions in lock.c and lock.h provide one way of achieving this mutual exclusion.

'int test_and_set(lock_t *flag);'
: Given a pointer to a lock flag, this function will atomically test and set the value pointed at (by using the 68000 tas instruction). It will return 0 if someone else already had the lock, and 1 if no one else already had the lock.

'void lock(lock_t *flag);'

'void unlock(lock_t *flag);'

Given a pointer to a lock flag, lock() will defer in a loop so long as someone else is holding the lock, and return when it has achieved the lock itself.

Unlock clears a lock, and should ALWAYS be used after lock() in all possible execution paths so that the next thread (or the same thread on another pass) can continue.

*** IMPORTANT ***

Lock should not be used by interrupt routines, as it runs the risk of leaving the processor running at too high an inerrupt level and blocking other interrupts which can mess things up. There is a bit of a safety net built in which prevents lock from blocking if the interrupt level is non-zero. However, this means that interrupt routines can break the invariants that lock was designed to protect – so just don't do it.

6.8 Interrupt functions

These functions affect the interrupt vector table directly.

'void install_exception_handler(int i,void *p);'

Installs the pointer p to an interrupt handler to correspond to interrupt i (in the range 0 to 256). After calling this function once to set up the vector, when the processor is interrupted with exception number i, it will call the handler pointed at by p.

The handler pointed to by p must be in assembler in order to preserve registers properly and execute the needed rte (return from exception) instruction instead of the default rts (return from subroutine) instruction.

Here is an example of a suitable interrupt handler – you should change the names of the functions, keeping in mind that C variable and function names have a prepended underscore when you refer to them in assembler, and vice versa (so "foo" in C becomes '_foo' in assembler, and '_bar' in assembler is 'bar' in C).

```
asm(".globl _my_handler
_my_handler:
     moveml #0xc0c0,sp@-   /* push registers d0, d1, a0, a1 */
     jsr    _my_c_handler   /* call your c handler */
     moveml sp@+,#0x0303   /* pop registers d0, d1, a0, a1 */
     rte
");
void my_c_handler()
{
  /* Do what you want to do on the interrupt here */
}
```

```
void main()
{
  install_exception_handler(INTERRUPT_NUM,my_handler);
  /* Now your exception handler is installed */
}
```

*** IMPORTANT ***

When user code exits, it is necessary to remove all exception vectors that point into that code and replace them with something safer. Therefore install_exception_handler() also keeps track of which exception vectors a process installs and removes them when the process exits. Keep this in mind and make sure that the process which calls install_exception_handler() is at least as long lived as any process that depends on that exception vector being set.

'`void *get_exception_vector(int i);`'
: Returns the pointer to the interrupt handler for interrupt i.

'`uint *get_vbr();`'
: You probably won't need to use this function because install_exception_handler() exists to protect you from it, but it will return to you the vector base register currently being used by the processor. The VBR is used during exception handling to know where the vector table starts so the correct interrupt handler can be used automatically. Be aware that with the way the system software is currently arranged, if you change the VBR you will likely be hurting yourself. Also, there is a default place that the system software puts the vector table and install_exception_handler() assumes that it doesn't move.

6.9 Multitasking

The multitasker takes care of process table management, process scheduling, crash handing, and time keeping (with time kept in swapper ticks). The scheduling interrupt on the 332 runs on the periodic interrupt so that it makes no assumptions about the hardware function allocation (in other words, it would be in many ways more convenient to use a TPU interrupt, but it would conflict with someone using all 16 timer lines for servos). The period of the scheduling interrupt is programmable. It is also possible for a process to give up the CPU before its allocated time is up by calling defer() or one of its derivatives. This uses a trap to cause an immediate swap to occur. Process scheduling is round robin. When the current process is preempted or defers the next process in the table is allowed to run.

I have been warned that I use the word "swap" incorrectly because swap implies that things are being put on disk. None of the 332 systems this software has yet been run on even have disks, so hopefully this potential confusion is not a problem. "Swap" is simply a more convenient term than "context switch" or anything else I can think of, so I will use it.

I have also been warned that there is a distinction between processes and threads. To many people, "processes" implies heavy-weight, virtual memory changing, belonging to different programs, UNIX style processes, whereas "threads" implies threads of execution which have shared memory space and come from the same program. Kernel processes are not exactly like either of these – all processes have shared memory space, and an arbitrary number of processes can be associated with a program.

There is a limit to the number of processes which can be active at a given time. Currently that limit is 32. I could change this if it ever becomes necessary.

The functions described in this section make up the user's interface to the multitasker.

6.9.1 Process creation/deletion

'`typedef void (*ps_func)();`'

'`pid start_process(const char *name,ps_func f,int stack_size,int time_slice);`'

'`pid start_adult_process(const char *name,ps_func f,int stack_size,int time_slice);`'

Both of these functions create and return the pid of a new process with the specified name, stack_size bytes of stack, time_slice scheduler ticks between being swapped in and being preempted (if it doesn't defer beforehand), and which is started with a call to function f (a void (void) function). The difference between start_process() and start_adult_process() is how they react to their parent being killed. start_adult_process() starts a process with a completely independent lifespan – it will live until function f exits or until it is killed. start_process() starts a standard child process which will die on the same conditions or when its parent dies, whichever comes first.

'`int kill_process(pid id);`'

kill_process() kills the process with the given pid. If the process is valid and killable (not root or gdb), kill_process() kills it and returns 1. Otherwise it does nothing and returns 0.

'`pid get_current_pid();`'

Returns the pid of the process currently executing.

'`pid lookup_pid_by_name(char *name);`'

Given a name, lookup_pid_by_name will either return the pid of the first process in the table with that name, or INVALID_PROCESS if it does not exist. This is the function called by the kill console command when it is given a string argument.

6.9.2 Scheduler control

'`int set_swap_tick_period(int ns);`'

'`int get_swap_tick_period();`'

set_swap_tick_period() sets the period of the scheduler interrupt to the number of nanoseconds specified and returns the period as actually set (which will probably differ due to the granular nature of the periodic interrupt setup). get_swap_tick_period() returns the current scheduler tick period.

'`int swap_cycle_time();`'

Returns the average number of scheduler ticks that it is taking to completely cycle through the process table. This number will range from 0 (if everyone is deferring) to the sum of the number of ticks alotted to all the processes (if no one is deferring). This number allows you to have an idea of the processor load.

'`void enable_process_swapping();`'

'`void disable_process_swapping();`'

These two functions allow the scheduler interrupt to be enabled and disabled. Disabling the interrupt allows CPU-hungry applications, such as software frame grabbing, to proceed uninterrupted. However, be aware that while process swapping is disabled the time will not increment so by using these functions you are accepting that you will have inconsistent time.

6.9.3 Defer

'`void defer();`'

defer() allows a process to give up the rest of its time and allow the next process in the cycle to be swapped in right away. defer() uses a trap to achieve this, as successful context switching requires the use of an rte instruction to get the status register right. Be aware that defer() has no effect on when the next regularly spaced scheduler tick will fall. Therefore, after a defer the next tick will be short. Therefore, if you care about a process getting a certain amount of work done before it is preempted you should let it have at least 2 ticks because the first one may not be complete.

'void defer_on_input_pending(FILE *f);'

'defer_on_input_pending()' marks the current process as an io_waiting process and it will not be swapped in until there is something in the input buffer of file f. This is automatically called by getchar() on an empty input buffer.

'void defer_until(int endtime);'

'defer_until()' marks the current process as a sleeping process and it will not be swapped in until the time is greater than or equal to endtime. This is automatically called by sleep() when the duration of the sleep exceeds the number of ticks left to the process.

'void hog_processor();'

'hog_processor()' gives the current process 256 more scheduler ticks. This is useful if you want to be fairly confident that you can carry out a computation uninterrupted but do not want to prevent the steady march of time.

6.9.4 Manipulation of running processes

'int set_process_time_slice(pid id,int new_slice);'

Allows the time slice of a running process to be changed. Returns 1 if the pid was valid and -1 if the pid was invalid.

'int get_process_time_slice(pid id);'

Returns the current time slice for a process, or -1 if the pid is invalid.

'FILE *get_process_instream(pid id);'

Returns the input stream of a process so that another process could talk to it over its standard text input stream if it wanted to.

6.9.5 Time functions

'int time();'

Returns the current time;

'int sleep(int counts);'

Sleeps for an amount of time greater than or equal to counts scheduler ticks. The sleeping time may not be anywhere near accurate if someone is turning off scheduler interrupts, and may be larger by an amount less than or equal to swap_cycle_time().

6.9.6 Printing functions

'`void print_process_regs(pid id,FILE *s);`'

Print the registers of the given process to the given stream.

'`void print_process_table();`'

Print the process table. This is what is called by the ps console command. An example of the output from this function follows:

```
Multitasking:
swap cycle time = 1, expected = 16, load average = 6%
PID     Status  Ticks  Low     Water   High    PC      Name
0       Running 5      13f800  13fd10  140000  702     Root process
33      IO Wait 10     130ca4  131334  1314a4  50d0    Debug Process
66      Crashed 1      1315e8  1318b0  1319e8  118144  Dummy process
```

6.10 332 hardware registers

The hardware registers for the 332 are memory mapped as bitfield structures with the same names as in the Motorola documentation. See the file mem.h in the kernel subdirectory of the main 332 tree for details. Basically you have the following pointers to the main blocks of 332 hardware registers, which work in both your programs and in gdb:

'`simptr sim;`'

Pointer to the system integration module memory map. Some examples are: sim->syncr - the clock control register sim->porte - the data byte for port E sim->picr.piv - the periodic control register interrupt vector number

Generally, if you are tempted to use these registers in your programs you might want to check sys.h and sys.c to see if there is a function that already does what you want.

'`csmptr csm;`'

Pointer to the chip select module memory map. Some examples are: csm->boot - the complete chip select settings for CSBOOT csm->csch[3].dsack - the data strobe acknowdge delay for CS3

Functions for setting up chip selects and changing their dsack delay can be found in sys.h and sys.c.

'`qsmptr qsm;`'

Pointer to the queued serial module memory map. This includes the control and data registers for the queued serial module and the asynchronous serial module both. Some examples are: qsm->tmcr.qmcr.iarb - the interrupt arbitration value for qsm qsm->scbr - the asynchronous serial baud rate conrol register

Functions for controlling the asynchronous serial module can be found in sys.h, sys.c, stdio.h, and stdio.c. The queued serial module is not curently built into kernel in any way. This is a planned improvement.

'tpu_control_ptr tpu_control;'

Pointer to the timer processor unit configuration and control registers. Some examples are: tpu_control->cier - channel interrupt enable register tpu_control->tmcr.stop - bit for stopping the entire TPU

Functions for controlling the TPU are found in TPU.h and TPU.c.

Actual memory map structures:

```
typedef volatile struct simmap {
  word mcr;
  word simtr;
  struct syncr_struct {
    uint w :1;
    uint x :1;
    uint y :6;
    uint ediv :1;
    uint   :2;
    volatile uint slimp :1;
    volatile uint slock :1;
    uint rsten :1;
    uint stsim :1;
    uint stext :1;} syncr;
  byte u1;  byte rsr;
  word simtre;
  uint u2;

  word u3;
  byte u4;   volatile byte porte;
  byte u5;   volatile byte eport;
  byte u6;   byte ddre;
  byte u7;   byte pepar;
  byte u8;   volatile byte portf;
  byte u9;   volatile byte fport;
  byte u10;   byte ddrf;
  byte u11;   byte pfpar;
  struct sypcr_struct {
     uint u12 :8;

     uint swe :1;                      /* watchdog enable */
     uint swp :1;                      /* watchdog prescale */
     uint swt :2;                      /* watchdog timing */
```

```
     uint hme :1;                        /* halt monitor enable */
     uint bme :1;                        /* bus monitor enable */
     uint bmt :2;                        /* bus monitor timing */
  } sypcr;                              /* system protection ctrl register */
  struct picr_struct {
    uint     :5;
    uint pirq:3;
    uint piv :8;} picr;
  struct pitr_struct {
    uint     :7;
    uint ptp :1;
    uint pitr:8;} pitr;
  byte u13;   byte swsr;
  word ua28;
  word ua2a;
  word ua2c;
  word ua2e;
  word tstmsra; /* a30 */
  word tstmsrb;
  byte tstsca;   byte tstscb;
  word tstrc;
  word creg;
  word dreg;
  uint u15;
  byte u16;   byte cspdr;} *simptr;

extern const simptr sim;

struct chip_select_channel {
    uint addr :13;
    uint blksz :3;
    uint mode :1;
    uint byte :2;
    uint rw :2;
    uint strb :1;
    uint dsack :4;
    uint space :2;
    uint ipl :3;
    uint avec :1;};

typedef struct chip_select_map {
  struct cspar_struct {
    uint   :2;
    uint cs5 :2;
    uint cs4 :2;
    uint cs3 :2;
    uint cs2 :2;
    uint cs1 :2;
```

```
    uint swsr1 :2;
    uint swsr2 :2;  /* end of first word */

    uint   :6;
    uint cs10 :2;
    uint cs9 :2;
    uint cs8 :2;
    uint cs7 :2;
    uint cs6 :2;} cspar;

  struct chip_select_channel boot;
  struct chip_select_channel csch[10];} *csmptr;

extern const csmptr csm;

struct synch_ser_comnd {
  uint cont   :1;
  uint bitse  :1;
  uint dt     :1;
  uint dsck   :1;
  uint psc3   :1;
  uint psc2   :1;
  uint psc1   :1;
  uint psc0ss :1;};

typedef volatile struct queued_serial_module {
  struct qmcr_struct {
    uint stop :1;
    uint frz1 :1;
    uint frz0 :1;
    uint      :5;
    uint supv :1;
    uint      :3;
    uint iarb :4;} qmcr;
  word qtest;
  struct qilr_struct {
    uint        :2;
    uint ilqspi :3;
    uint ilsci  :3;
    uint qivr   :8;} qir;
  word r0;
  word scbr;  /* baud control register */
  struct sccr_struct {
    uint    :1;
    uint loops :1;
    uint woms  :1;
    uint ilt   :1;
    uint pt    :1;
```

```
    uint pe    :1;
    uint m     :1;
    uint wake  :1;
    uint tie   :1;
    uint tcie  :1;
    uint rie   :1;
    uint ilie  :1;
    uint te    :1;
    uint re    :1;
    uint rwu   :1;
    uint sbk   :1;} sccr;
  volatile struct scsr_struct {
    uint      :7;
    volatile uint tdre :1;
    volatile uint tc   :1;
    volatile uint rdrf :1;
    volatile uint raf  :1;
    volatile uint idle :1;
    volatile uint or   :1;
    volatile uint nf   :1;
    volatile uint fe   :1;
    volatile uint pf   :1;} scsr;
  volatile word scdr;
  uint r1;
  byte r2;    byte qpdr;
  union qpar_qddr_union {
    struct qpar_struct {
      uint       :1;
      uint pcs  :3;
      uint pcs0ss :1;
      uint       :1;
      uint mosi  :1;
      uint miso  :1;
      uint       :8;} qpar;
    struct qddr_struct {
      uint       :8;
      uint txd   :1;
      uint pcs   :3;
      uint pcs0ss :1;
      uint sck   :1;
      uint mosi  :1;
      uint miso  :1;} qddr;
  } qpar_qddr;
  struct spcr0_struct {
    uint mstr    :1;
    uint womq    :1;
    uint bits    :4;
    uint cpol    :1;
```

```
    uint cpha    :1;
    uint spbr    :8;} spcr0;
  struct spcr1_struct {
    uint spe     :1;
    uint dsckl   :7;
    uint dtl     :8;} spcr1;
  struct spcr2_struct {
    uint spifie  :1;
    uint wren    :1;
    uint wrto    :1;
    uint         :1;
    uint endqp   :4;
    uint         :4;
    uint newqp   :4;} spcr2;
  struct spcr3_spsr_struct {
      uint       :5;
      uint loopq :1;
      uint hmie  :1;
      uint halt  :1;
      uint spif  :1;
      uint modf  :1;
      uint halta :1;
      uint       :1;
      uint cptqp :4;} spcr3_spsr;
  byte r3[0xe0];
  word rec_ram[16];
  word tran_ram[16];
  byte comd_ram[16];} *qsmptr;

extern const qsmptr qsm;

typedef struct tpu_control_struct {
  struct tmcr_struct {
    uint stop :1;
    uint tcr1_prescale :2;
    uint tcr2_prescale :2;
    uint emu :1;
    uint t2cg :1;
    uint stf :1;
    uint supv :1;
    uint psck :1;
    uint      :2;
    uint interrupt_arbitration :4;} tmcr;
  word ttcr;
  word dscr;
  word dssr;
  struct ticr_struct {
    uint      :5;
```

```
      uint interrupt_request_level :3;
      uint interrupt_base_vector :8;} ticr;
    word cier;
    word cfs[4];
    word hsf[2];
    volatile word hsrf[2];
    word cpr[2];
    volatile word cisr;} *tpu_control_ptr;

  extern const tpu_control_ptr tpu_control;
```

6.11 332 System Integration Module functions

The

6.11.1 General Use functions

'`int period_of(int hz);`'

Given rate in hertz, period_of returns period in ns

'`int rate_of(int ns);`'

Given period in nanoseconds, rate_of returns rate in hertz

6.11.2 Digital I/O support

'`void set_porte_output(byte mask);`'

'`void set_porte_input(byte mask);`'

Sets port E lines masked as 1s in argument to output/input

'`int read_porte();`'

Returns the current value on port E data pins. The bits only have meaning if they are configured to port E input.

'`void write_porte(byte data);`'

Writes the data to port E data pins. Only has effect for bits configured for port E output

'`void set_porte_bus_control(byte mask);`'

Sets masked port E lines to alternate bus control functions '`void set_portf_output(byte mask);`'

'`void set_portf_input(byte mask);`'

Sets port F lines masked as 1s in argument to output/input

'`int read_portf();`'

Returns the current value on port F data pins. The bits only have meaning if they are configured to port F input.

'`void write_portf(byte data);`'

Writes the data to port F data pins. Only has effect for bits configured for port F output

'`void set_portf_interrupt_request(byte mask);`'

Sets masked port F lines to alternate interrupt functions

'`** In theory, the port C functions should work, but I have not really`'

tested them yet. Port C coexists with the chip selects, so be very careful in using these functions, as bashing the chip selects on the ROM or RAM will crash you right away.

'`void set_portc_output(byte mask);`'

Sets masked port C lines to output function. The MSB when configured for output becomes the Eclock. The other bits are normally chip selects and can be configured for discrete output.

'`void write_portc(byte data);`'

Writes the data to port C data pins. Only has effect for bits configured for port C output and never for the MSB.

6.11.3 Clock control

'`int clock_rate();`'

Returns the current system clock rate in hertz. This function is memoized so calculation is not usually necessary so don't stress about using this function.

'`int clock_period();`'

Returns the current system clock period in nanoseconds.

'`int set_clock_rate(int speed);`'

Set_clock_rate takes an argument in hertz, sets the clock to the closest possible approximation of that speed, and returns the clock rate actually set (in hertz).

'`void wait_clock_lock();`'

This probably isn't practical, but after a clock rate change this will wait until the clock source has locked onto the new speed after a clock speed change.

6.11.4 Chip select module

```
enum cs_pin {output,default_config,cs_8bit,cs_16bit};
enum cs_size {s2k,s8k,s16k,s64k,s128k,s256k,s512k,s1m};
enum cs_mode {async,sync};
enum cs_byte {off,lower,upper,both};
enum cs_rw {read=1,write,rw};
enum cs_strobe {as,ds};
enum cs_space {cpu,user,supervisor,u_s};
enum cs_avec {avec_off,avec_on};
```

'`int initialize_chip_select(int num,enum cs_pin pin,void *base_addr,`'

enum cs_size size,enum cs_mode mode, enum cs_byte byte_config, enum cs_rw rw,enum cs_strobe strobe, byte dsack_delay,enum cs_space space, byte interrupt_level, enum cs_avec avec); Configures the chip select num to the options provided. Num may range from 0 to 10 (returns -1 if out of range), but on the Motorola bcc board uses chip selects 0, 1 and 2 and the Vesta board uses chip selects 0 and 1 for main ram. The cs_pin argument should only be either cs_8bit for 8 bit transfers or cs_16bit for 16 bit transfers. Dsack_delay can be from 0 to 13 wait states per access (with 14 meaning it can go real fast and 15 meaning external ack), and interrupt_level from 1 to 7 (with 0 meaning all interrupts allowed). The other parameters should be self explanatory from the enums listed above, and for details of operation refer to the 68332 user's manual pages 4-27 to 4-45. The format of this instruction corresponds directly to the format of table 4-18 on page 4-38 of the text.

After configuring a chip select using this function, any reference to the block of memory starting at the base_address (which must lie on an appropriate boundary) and the block of memory the size of which is specified by size will cause the appropriate chip select to give your peripheral an active low chip select signal telling it to either drive the bus if the operation is a read or take data from the bus if the operation is a write. This effectively memory maps your external device for you!

'`void set_dsack(int cs,int val);`'

This function will set the dsack for the specified chip select to the specified number of wait states. See above for description of the numbers.

'`int set_cs_size(int cs,int val);`'

Assuming that the chip select cs is already configured, this function will set the block size for the chip select to the smallest block size greater than or equal to the length in bytes specified by val. The value returned is the actual length in bytes that the chip select was set to. Val should be a power of 2, but enven then only a limited set of lengths are possible.

The possible lengths and their representations in hexidecimal are as follows:

```
2k = 0x800        16k = 0x4000   128k = 0x20000   512k = 0x80000
8k = 0x2000       64k = 0x10000  256k = 0x40000   1M   = 0x100000
```

6.11.5 Asynchronous Serial configuration

'`int baud_rate();`'
: Returns the current baud rate of the 68332.

'`int set_baud_rate(int baud);`'
: Given a nominal baud rate to shoot for, this will set the asynchronous baud rate to as close to that as possible and return the exact baud as set (usually some error necessary).

'`int recalibrate_baud();`'
: You should not ever need to call this as you should only change the clock speed by calling set_clock_rate which does this automatically, but this function will recalibrate the system to maintain a stable baud rate despite changes in system clock speed.

6.12 332 TPU functions

This module includes functions for affecting the Timer Processor Unit as a whole, and general functions for affecting the actions of each pin.

For all the functions which take a channel number as an argument, the channel number should be in the range 0 to 15, which correspond directly with the TPU pin number. Therefore pin TPU0 corresponds to an argument 0, TPU15 to 15, etc. This argument could also be channel_of(ptr) where ptr is the handle returned from an initialization function on a channel.

'`#define channel_of(ptr) {blah...}`'
: Given a pointer to the parameter space of a TPU function (which is what the initialization functions return) this will tell you what channel that pointer corresponds to.

Here is the intended use: If you had a pwm channel that you initialized by:

```
pwm pptr=pwm_init(blah blah);
```

that you wanted to disable temporarily (or call any of the general TPU functions that require a channel as an argument), you could use:

```
set_timer_priority(channel_of(pptr),0);
```

and not have to keep track of what channel it was initialized on. This operation is quite efficient (it only requires bit masking, not any sort of table lookup), and I recommend using it for better code readability.

6.12.1 TPU module configuration

These functions affect the configuration and operation of the TPU as a whole.

'`void stop_tpu();`'
: Stops the TPU. No channels are serviced. The TCRs don't increment. It's dead, Jim.

'`int tpu_stopped();`'
: Returns 1 if TPU is stopped, 0 if not.

'`void start_tpu();`'
: Restarts the TPU after it has been stopped.

'`void initialize_tpu_interrupts();`'
: Calling this function configures the TPU to allow it to trigger CPU interrupts. This function is automatically called by kernel, so you should not need to call it.

'`void disable_tpu_interrupts();`'
: This will uniformly prevent the TPU interrupts from being acknowledged by the CPU in case you don't want it to bother you for some amount of time...

'`void enable_tpu_interrupts();`'
: This will undo the effect of the previous function and reenable TPU interrupts

6.12.2 TCR control

The time base for the TPU channel functions can either be TCR1 or TCR2. Both are 16-bit free-running counters. TCR1 is always a divisor of the system clock. TCR2 is affected by the pin T2CLK. These functions affect the TCRs.

```
enum psck_enum {psck_div32,psck_div4};
enum tcr_prescale {div1,div2,div4,div8};
enum tcr2_pin_use {pin_is_clock_source,pin_is_gate_of_div8_clock};
typedef enum tcr_source_enum {tcr1= 1, tcr2= 2} tcr_source;
```

'`int initialize_tpu_speeds(enum psck_enum psck,enum tcr_prescale t1p,enum tcr_prescale t2p);`'

It is possible to set the divisor from the system clock (psck), the prescales from that clock source to TCR1, and the prescale for TCR2 **ONLY ONCE BETWEEN RESETS**. Worse than that, the same register that controls the the prescales also controls the interrupt arbitration id, which must be set during kernel initialization. Therefore, the first time the user try to change the prescales it will not work.

The most recently requested prescale values are stored in persistents, and at the next reset the values will be set correctly. initialize_tpu_speeds will return 1 if the values actually in the register match the requested values, and 0 if they don't.

The best thing to do if this returns 0 is to request that the user hit reset, and it will be set right the next time. The system software specifically does not try to set the prescales, and you should make sure that your code either tries to set it only once, or that all places which try to set the prescales agree on what they should be. YOU SPECIFICALLY *CANNOT* CHANGE THESE VALUES DYNAMICALLY.

The following table gives the results of setting the prescales for a 16.7MHz system clock. If the system clock is a different speed, scale these numbers accordingly.

```
TCR1 prescale |  psck=psck_div32       |  psck=psck_div4        |
              |div from sys | speed    | div from sys | speed   |
    div1      |     32  **  |  2us     |      4       | 250 ns  |
    div2      |     64      |  4us     |      8       | 500 ns  |
    div4      |    128      |  8us     |     16       |   1 us  |
    div8      |    256      | 16us     |     32       |   2 us  |

TCR2 prescale | divide by | div from sys | div from external    |
    div1      |       1 **|       8      |          1           |
    div2      |       2   |      16      |          2           |
    div4      |       4   |      32      |          4           |
    div8      |       8   |      64      |          8           |
```

'`void set_tcr2_source(enum tcr2_pin_use);`'

- pin_is_clock_source

  Sets the source for TCR2 to be the signal at the pin T2CLK.

- pin_is_gate_of_div8_clock

  Sets the source for TCR2 to be the system clock divided by 8 and gated by the pin. In other words, if the system clock is 16MHz and you want TCR2 to go at 2MHz, use this option and tie T2CLK to VCC. If you want TCR2 to accumulate pulse widths of a signal, hook that signal to TCR2 and the TCR2 count * .5us is the accumulated time.

'`int calculate_tcr1_speed();`'
: You shouldn't need to call this as it is automatically called if you change the clock speed or prescale using the functions provided. However, it pre-calculates and stores the period and rate of TCR1 and returns the rate in hertz.

'`int tcr1_rate();`'
: Returns TCR1 speed in hertz.

'`int tcr1_period();`'
: Returns TCR1 period in nanoseconds.

'`int period_to_tcr1_counts(int period,int *error);`'
: Returns the number of TCR1 counts that comes closest to while not exceeding the period argument in nanoseconds and stores the error in nanoseconds of the resulting value in *error. TCR1 is only a 16-bit counter, so you should check that the value returned is less than 65000 before using it for TPU functions. If the value is too large, consider setting the TCR1 prescale and/or PSCK to values that cause TCR1 to run slower. See '`initialize_tpu_speeds()`'.

'`int rate_to_tcr1_counts(int hertz,int *error);`'
: Same as above, but argument is in hertz. Error is still in nanoseconds. The same warnings about overrunning 16 bits apply as above.

'`int tcr1_counts_to_period(int counts);`'
: Returns the period in nanoseconds corresponding to the given number of TCR1 counts.

'`int tcr1_counts_to_rate(int counts);`'
: Returns the rate in hertz corresponding to the given number of TCR1 counts.

6.12.3 Channel control

All built-in TPU functions require the channel control field to be set as part of channel initialization. The channel control structure actually consists of three fields which affect the TPU hardware latches associated with each pin – pin state control, pin action control, and time base/directionality seletion.

The actual types used to map on to the channel control field of the TPU register are as follows. The reason a union is used rather than just a plain struct is that you may want to load all the fields at once for efficiency (and therefore use the '`w`' part), or you may want to load each field inividually without worrying about shifts and masks (and therefore use the '`bf`' part).

```
typedef union {
  word w;
```

```
    struct {
      uint    :7;
      uint tbs:4;
      uint pac:3;
      uint psc:2;} bf;} channel_control;

  enum pac_input {no_detect_edge,rising_edge,falling_edge,either_edge};
  enum tbs_input {i_c1_m1,i_c1_m2,i_c2_m1,i_c2_m2};

  enum psc_output {follow_PAC,force_high,force_low,no_force};
  enum pac_output {no_change,high_on_match,low_on_match,
                   toggle_on_match,preserve_pac};
  enum tbs_output {o_c1_m1=4,o_c1_m2,o_c2_m1,o_c2_m2,preserve_tbs};
```

Pin State Control (psc)

Controls the immediate action of a pin in response to initialization. This field only affects output channels (and therefore is not even an argument to the function to set the channel control of an input). The values are as follows:

'`follow_PAC`'
: Do what the Pin action control field specifies

'`force_high`'
: Force output high on initialization

'`force_low`'
: Force output low on initialization

'`no_force`'
: Leave the pin state as you found it

Pin Action Control:

Controls the action of a pin in response to a match or transition. The meaning of this field is affected by whether the pin is an input or output.

- Input

  '`no_detect_edge`'
  : Do not do anything when a transition occurs

'rising_edge'
: Capture rising edges

'falling_edge'
: Capture falling edges

'either_edge'
: Capture rising and falling edges

'preserve_pac'
: Leave the PAC at the value it had last time

- Output

'no_change'
: Do not change the output level when a match occurs

'high_on_match'
: Set the output to be high when a match occurs

'low_on_match'
: Set the output to be low when a match occurs

'toggle_on_match'
: Toggle the output when a match occurs

'preserve_pac'
: Leave the PAC at the value it had last time

Time Base/Directionality Selection (tbs)

Controls whether the channel is input or output, and whether it uses TCR1 or TCR2. It is actually possible to match on and capture different TCRs, but in general you want to use the same TCR for both.

- Input

'i_c1_m1' Input capture TCR1, match TCR1

'i_c1_m2' Input capture TCR1, match TCR2

'i_c2_m1' Input capture TCR2, match TCR1

'i_c2_m2' Input capture TCR2, match TCR2

'preserve_tbs'
: Leave TBS as it was

- Output

'o_c1_m1' Output capture TCR1, match TCR1

'o_c1_m2' Output capture TCR1, match TCR2

'o_c2_m1' Output capture TCR2, match TCR1

'o_c2_m2' Output capture TCR2, match TCR2

'preserve_tbs'
Leave TBS as it was

Channel Control functions

'channel_control channel_control_input(enum pac_input pac,enum tbs_input tbs);'

'channel_control channel_control_output(enum psc_output psc,enum pac_output pac,enum tbs_output tbs);'

These functions return the value of the channel_control parameter required for initializing TPU functions. These functions take their arguments and pack them appropriately to insert into the TPU parameter ram or pass to channel initialization functions. Example of using these functions follow:

- Input

```
channel_control ctl=channel_control_input(either_edge,i_c1_m1);
din=dio_input_init(ch,100,transition,3,ctl);
```

- Output:

```
channel_control ctl =
  channel_control_output(force_low,toggle_on_match,o_c1_m1);

sqwave=pwm_init(15,1000,2000,1,ctl);
```

6.12.4 TPU Interrupt support

'void install_tpu_exception_handler(int ch,void *p);'

This function installs the given function as the interrupt handler for the given channel. This DOES NOT enable the interrupts on the channel and should be called before enabling the interrupts on a channel.

'void set_timer_interrupt_enable(int ch,int val);'

This function enables or disables interrupts on the given TPU channel. If val==0 the given channel will not interrupt the CPU, when an interrupt-inducing event occurs (transition, match, whatever), but will instead just set a bit in the timer interrupt status field (which can be polled using timer_interrupt_status). If val==1, then exception handler previously installed by a call to install_tpu_exception_handler() will be called when an interrupt occurs.

'`int timer_interrupt_status(int ch);`'
: This is useless if the interrupts are enabled on the given channel. If interrupts are disabled, timer_interrupt_status will return 0 if no interrupt has occurred, or 1 if an interrupt has occured. After reading that an interrupt has occurred, you must clear the status flag by a call to reset_timer_interrupt. Interrupts are disabled by default, or by calling set_timer_interrupts_enable(ch,0) at any time.

'`void reset_timer_interrupt(int ch);`'
: If interrupts are disabled, use this to reset the interrupt status bit after to allow polling for a new interrupt. If interrupts are enabled, you can call this from your interrupt routine to reset for the next interrupt. Note that if you are using intstall_tpu_exception_handler, you do not need to call this in your interrupt as the status bit is reset automatically. However, it doesn't hurt to call it anyway. If you are using your own interrupt wrapper, you should call definitely call this, as failing to will cause the interrupt to be repeatedly handled and prevent other processing from happening.

'`int wait_for_timer_interrupt(int channel);`'
: *** Only use this function on channels for which interrupts are disabled!! ***** This function will loop until the status bit of the given channel is set, then it will reset the status bit and return.

6.12.5 General timer channel configuration

'`void set_timer_priority(int ch,int val);`'
: Sets the timer priority for the given channel to val where val is in the range of 0 (disable service to the channel) to 3 (this is a really important channel, service promptly). See the Motorola TPU reference manual for more details about how priority affects service latency.

'`int wait_for_host_service_complete(int channel);`'
: The CPU communicates with the TPU through shared parameter space. Therefore when the CPU requests service from the TPU there may be some delay before the request is completed. Call this function if you need to wait until a service request has been completed before continuing with program execution. In most cases this is not necessary. It is only necessary when synchronization between the CPU and TPU is required.

6.12.6 Channel functions

Please note the following general things about using channel functions:

- The channel_control argument to all channel functions should be created using channel_control_input() or channel_control_output(). See the section on channel control above for details.
- The channel control field of a channel functions parameter space is only used during initialization. If you want to change the channel control (for instance, change a rising-edge channel into a falling- edge channel), you MUST reinitialize the channel, or at least issue a host service request.
- The channel argument (ch) must always be in the range of 0 to 15.
- If a channel that was previously configured for one function is subsequently re-initialized, only the most recent initialization matters.
- The pointers (or you could perhaps better think of them as objects) returned by the initialization functions can be used as a pointer into the parameter space for the channel. For example, if foo is an object of type dio returned by the dio initialization function, then `foo->pin_level` is the pin level of that tpu channel.
- You can recover the channel from a tpu function pointer by using `channel_of(ptr)`.
- Priority is a number from 0 (lowest priority=disable) to 3 (highest priority) that is used to arbitrate simultaneous servicing requests.
- If you wish to enable interrupts on a channel, the appropriate ordering of the initialization steps is as follows:
  - Initialize channel using an initialization function like `pwm_init`,`itc_init`, `dio_output_init`, etc.
  - Call install_tpu_exception_handler to specify which function to execute on interrupt.
  - Call set_timer_interrupt_enable when you want to enable interrupt.

  Executing these steps out of order can lead to serious problems.

6.12.6.1 DIO

To use these functions you must '`#include "tpu/dio.h"`'.

```
typedef struct dio_struct {
  channel_control ctl;
  volatile word pin_level;
  word match_rate;} *dio;

enum dio_hsf {transition=0,match=1,cpu_sample=3};
```

'`dio dio_output_init(int ch,int state);`'

Initializes the channel to discrete output with initial value state (either 1=HIGH or 0=LOW). Subsequent changes to the output level can be made by calling dio_out on the initialized channel.

'`void dio_out(dio dptr,int state);`'

Set pre-initialized discrete output channel to given state.

'`dio dio_input_init(int ch,int match_rate,enum dio_hsf hsf,int priority,channel_control ctl);`'

Initializes the channel to discrete input, updating either on input transition, a a constant rate match_rate (this parameter is meaningless if hsf!=match...), or when the cpu requests update.

'`int dio_state(dio dptr);`'

Returns a word (lower two bytes of int...) corresponding to the last 16 states of the input channel.

'`int dio_now(dio dptr);`'

Returns 0 if current (or most recently sampled) discrete input state is low and non-zero (actually 0x8000 'cause it's cheap) otherwise.

6.12.6.2 PWM

To use these functions you must '`#include "tpu/pwm.h"`'.

```
enum pwm_hsrf {pwm_initialize=2,pwm_update=1};

typedef struct pwm_struct {
  channel_control ctl;
  volatile word oldris;
  word pwmhi;
  word pwmper;
  volatile word pwmris;
  word slate;} *pwm;
```

'`pwm pwm_init(int ch,int hightime,int period,int priority,channel_control ctl);`'

Initializes the channel to pulse width modulated output with the given period and hightime (where hightime/period=duty cycle of the wave).

'`void pwm_set_times(pwm pptr,int high,int per);`'

Changes the high time and period on an initialized pwm channel to the given values.

'`void pwm_disable(pwm pptr);`'

'`void pwm_enable(pwm pptr,int priority);`'
: pwm_disable really just sets the channel priority regiter to 0, but results in suspending the output of a pwm channel without otherwise disturbing it. pwm_enable undoes this hold and assigns the channel the given priority.

6.12.6.3 ITC

To use these functions you must '`#include "tpu/itc.h"`'.

```
typedef struct itc_struct {
  channel_control ctl;
  volatile word lb;
  word max_count;
  volatile word trans_count;
  volatile word final_trans_time;
  volatile word last_trans_time;} *itc;

enum itc_hsf {single_no_links,continual_no_links,
              single_links,continual_links};
```

'`itc itc_init(int ch,int max_cnt,enum itc_hsf hsf,int priority,channel_control ctl);`'
: Initializes a channel for input timer capture/transition count. Note that links are not yet supported in this version of the libraries, so the valid values of hsf are single_no_links for one-shot action and continual_no_links for continuous action.

'`void set_itc_max_cnt(itc iptr,int cnt);`'
: Allows the maximum number of counts for an initialized itc channel to be reset.

'`int get_itc_last_period(itc iptr);`'
: Returns the last period accumulated by an initialized itc channel.

'`int reset_itc_counts(itc iptr);`'
: Resets the number of counts an itc channel has accumulated

6.12.6.4 OC

To use these functions '`#include "tpu/oc.h"`'.

This is unbelievably complex. I don't fully understand it yet. I'll document output compare when I understand it...

```
typedef struct oc_struct {
  channel_control ctl;
  word offset;
  word rat_ref1;
  word ref2_3;
  volatile word ref_time;
  volatile word actual_match_time;} *oc;

enum oc_hsrf {host_match=1,continuous_pulse=3};
enum oc_hsf {all=0,TCR_only=2};

#define TCR1 0xec
#define TCR2 0xee
#define TCR1_time() ((int)*(word *)0xffffec)
#define TCR2_time() ((int)*(word *)0xffffee)
#define timeptr_of(p) (((word)(p)) & 0xff)

oc oc_init(int ch,word timeptr,int priority,channel_control ctl);

oc oc_pulse_init(int ch,word timeptr,int width,int priority);

void oc_initiate(oc optr,int width);
int oc_timecalc(oc optr,int timeptr);
```

6.12.6.5 PPWA

To use these functions you must '`#include "tpu/ppwa.h"`'

```
typedef struct ppwa_struct {
  channel_control ctl;
  word max_period_cnt;
  volatile word last_accum;
  volatile word accum;
  volatile word accum_rate_ub;
  volatile word ppwa_lw;} *ppwa;

enum ppwa_hsf {period_24=0,period_16,pulse_24,pulse_16};
```

'`ppwa ppwa_init(int ch,int max_counts,int accum_rate,int priority,enum ppwa_hsf hsf,channel_control ctl);`'

Initializes the channel to execute pulse width or period accumulation for max_counts waveforms between interrupts and place the finished results in its parameter space. For consistency, if initializing ppwa channels for period measurement, the pac field of channel_control should be to detect falling edges and for pulse width measurement it should be to detect rising edges.

'void ppwa_set_accum_rate(ppwa pptr,int val);'
Allows resetting of the accum_rate for an initialized ppwa channel.

'void ppwa_reset_high_byte(ppwa pptr);'
For 24bit measurement, the high byte does not automatically get reset so you can use this function to do that.

'void ppwa_set_max_counts(ppwa pptr,int val);'
Allows resetting of the max_count for an initialized ppwa channel.

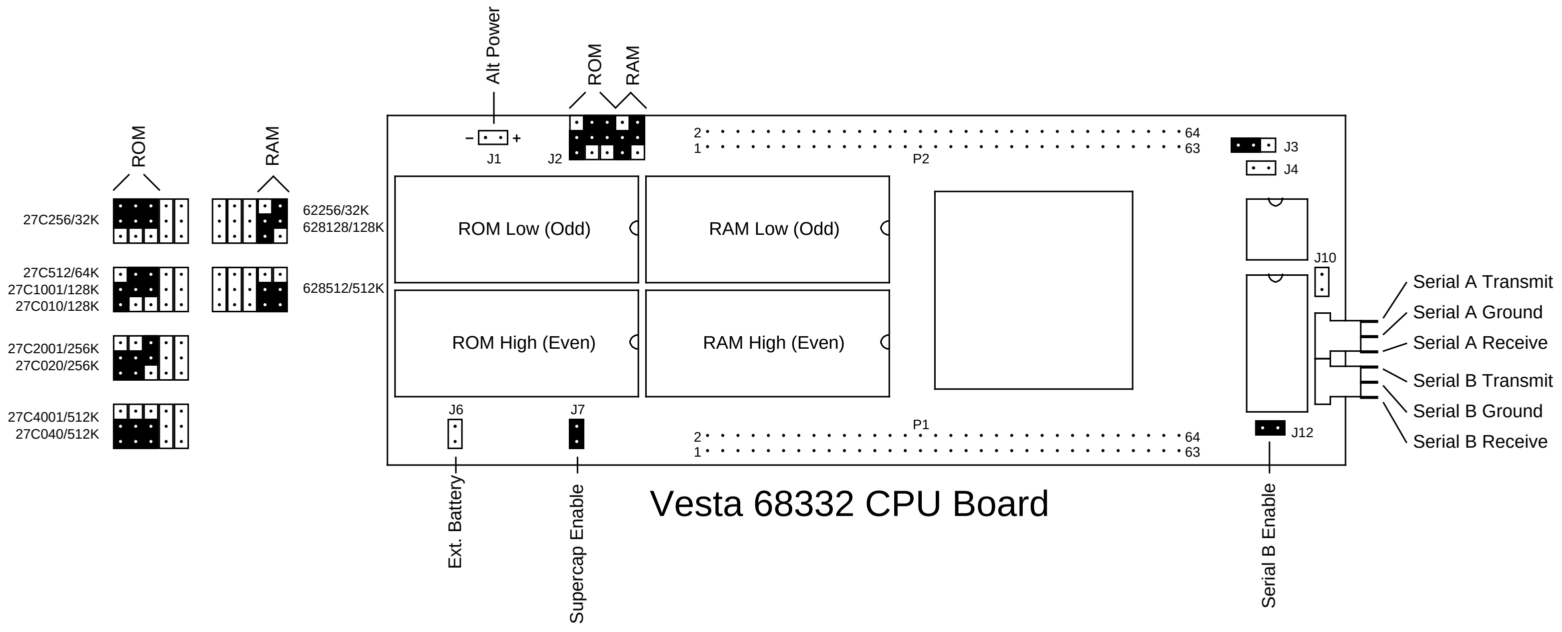
'int ppwa_full_period(ppwa pptr);'
Returns the full period for 24bit accumulation. This function does not, however, clear the upper byte of the accumulator so be careful to use this in concert with ppwa_reset_high_byte().

'int ppwa_ave_period(ppwa pptr);'
Divides the accumulated periods/pulse widths by the number of counts it represents and returns that value (rounded to an integer...).

'int ppwa_last_pulse_time(ppwa pptr);'
Returns the last TCR1 time at which a pulse was detected.

7 Vesta Hardware Specifics

7.1 Vesta Board Diagram



7.2 Vesta RAM/ROM selection Jumpers

The Vesta board works with either 28 or 32 pin ROMs and RAMs. The cost of this is that you have to worry about the jumper settings on the RAM/ROM selection block.

Jumper block J2, as you look at it with J2 at the top, the rom on your left, and the 332 on your right, looks like this:

```
    |--EPROM--| |-RAM-|
     --- --- --- --- ---
  J |   |   |   |   |   |
  2 | O | O | O | O | O |
    |   |   |   |   |   |
    | O | O | O | O | O |
    |   |   |   |   |   |    RAM
    | O | O | O | O | O |     |
    |___|___|___|___|___|     V     68332 ->

  (J2 orientation)
```

ROM Jumper selection

Set the three leftmost jumpers according to the ROM size you are using:

32Kx8 EPROM (27C256)

```
     --- --- --- --- ---
  J |***|***|***|   |   |
  2 |*o*|*o*|*o*| O | O |
    |***|***|***|   |   |
    |*o*|*o*|*o*| O | O |
    |***|***|***|   |   |     RAM
    | O | O | O | O | O |      |
    |___|___|___|___|___|      V    68332 ->
```

64Kx8 EPROM (27C512), 128Kx8 EPROM (27C1001, 27C010)

```
     --- --- --- --- ---
  J |   |***|***|   |   |
```

```
2 | O |*o*|*o*| O | O |
  |***|***|***|   |   |
  |*o*|*o*|*o*| O | O |
  |***|***|***|   |   |     RAM
  |*o*| O | O | O | O |      |
  |***|___|___|___|___|      V    68332 ->
```

256Kx8 EPROM (27C2001, 27C020)

```
   ___ ___ ___ ___ ___
J |   |   |***|   |   |
2 | O | O |*o*| O | O |
  |***|***|***|   |   |
  |*o*|*o*|*o*| O | O |
  |***|***|***|   |   |     RAM
  |*o*|*o*| O | O | O |      |
  |***|***|___|___|___|      V    68332 ->
```

512Kx8 EPROM (27C4001, 27C040)

```
   ___ ___ ___ ___ ___
J |   |   |   |   |   |
2 | O | O | O | O | O |
  |***|***|***|   |   |
  |*o*|*o*|*o*| O | O |
  |***|***|***|   |   |     RAM
  |*o*|*o*|*o*| O | O |      |
  |***|***|***|___|___|      V    68332 ->
```

RAM Jumper selection

Set the two rightmost jumpers according to the RAM size you are using:

32Kx8 RAM (62256), 128Kx8 RAM (628128)

```
   ___ ___ ___ ___ ___
J |   |   |   |   |***|
2 | O | O | O | O |*o*|
  |   |   |   |***|***|
```

```
 | O | O | O |*o*|*o*|
 |   |   |   |***|***|     RAM
 | O | O | O |*o*| O |      |
 |___|___|___|***|___|      V    68332 ->
```

512Kx8 RAM (628128)

```
    ___ ___ ___ ___ ___
 J |   |   |   |   |   |
 2 | O | O | O | O | O |
   |   |   |   |***|***|
   | O | O | O |*o*|*o*|
   |   |   |   |***|***|     RAM
   | O | O | O |*o*|*o*|      |
   |___|___|___|***|***|      V    68332 ->

  512Kx8 RAM (628128)
```

7.3 Second serial port

Vesta boards come wired for a second serial port. TPU pins 1 and 4 are wired to extra receive and transmit lines respectively on the serial level converter chip. When dealing with serial ports using ARC, this port is `serialb` and the built-in uart `seriala`.

To enable the second serial port follow this procedure:

- Remove J10 (located beside the serial connectors) to enable the port regardless of what TPU5 is doing.
- Place a jumper across J12 to connect TPU1 to the receive line. (On more recently produced Vesta boards, this jumper may not exist).
- Connect a serial cable to pins 3, 4, and 5 of J5 which is just like the standard one connected to pins 0, 1, and 2.

Now you can use the second serial port.

The baud rate of the second serial port defaults to 9600 baud, but may be changed by setting the persistent `serialb_baud` to a different value and resetting the board.

To make portb be the default serial interaction port, set the persistent `default_port` to 1. The default value of 0 specifies that seriala should be used.

To open streams on portb, use '`fopen("/serialb/foo","")`' (where foo is replaced by whatever you want to name the stream) for multiplexed streams or `fopen("/serialb/foo","R")` for a raw unmultiplexed stream.

Appendix A Reaching Newton Labs

When you have questions about the ARC development system and this manual doesn't help, please call us. Currently, the primary contact for ARC related questions is Anne Wright (`anarch@newtonlabs.com`), although other people will be able to help you if your need is urgent.

Also, we greatly appreciate any feedback you might give us about the ARC Development System and this documentation.

Address:

Newton Research Labs
Evergreen Park
16504 NE 37th St. #X2073
Redmond, WA 98052
Voice: (206) 869-4988
Fax: (206) 882-2043
E-mail: support@newtonlabs.com
http://www.newtonlabs.com

Appendix B Building Serial Cables

This section describes how to interface your RS-232 device to the MIT RJ-11 serial standard. The MIT RJ-11 serial standard works for:

- All Newton Labs computer boards and robot controllers
- The MIT 6.270 robot controller
- The MIT Miniboard (versions 1.5 and 2.0)
- The "Rug Warrior" (from "Mobile Robots: Inspiration to Implementation", by Joe Jones and Anita Flynn)

Most people will want to start with the commonly available DB-25 and DB-9 to RJ-11 converters (generally available from DigiKey or Jameco). The part numbers for DigiKey are included at the end of this section. If you buy one that fits your computer, the rest of the guide will tell you how to connect its internals to match the MIT RJ-11 standard.

B.1 Serial ports on various machines

IBM PC and compatibles

IBM PC or compatible may have either a male DB-25 or a male DB-9 coming out the back. (A female DB-25 is typically a parallel printer port, while a female DB-9 is typically for an old EGA-style monitor).

Macintosh

Macintosh computers have two 8-pin mini DIN connectors on the back. One is labeled with a phone icon and is called a modem port. The other is labeled with a printer and is called a printer port. For our purposes, the hardware for these ports are identical.

The best way to hook up to a Macintosh is to buy a "Haye's compatible modem cable" which acts as a mini DIN to Male DB25 adapter. The Male DB25 works just like IBM and compatable DB25, and should be wired the same. Macintosh "Printer Cable"s are identical except that they have pins 2 and 3 swapped. (Often, Mac MiniDIN-8 to DB25's aren't labeled, so you won't know which you have. Either experiment, or try adding a null modem to make yours work).

SPARCstations

SPARCstation 1's have two female DB-25's coming out of the back of the box ('`/dev/ttya`' and '`/dev/ttyb`'). Use the instructions for making a phone to male DB25 converter below.

Other SPARCstations have only one female DB-25 for serial, labeled "A/B". The standard male DB25 adapter pinout will let you hook up to port "A" ('`/dev/ttya`'). Port B uses a disjoint set of pins on the same DB-25, so you'd probably want to make some sort of custom Y-adapter to get access to both ports at the same time. (This document doesn't currently contain the pins of this DB-25 used for port "B").

Still other SPARCstations have 8-pin mini DIN connectors (they are round with 8 small pins). These connectors are the same as Macintosh serial connectors, and the same instructions should be used.

VAXstations

The VAXstation 2000 has a male DB-9 coming out the back ('`/dev/ttys1`'). *WARNING* This is a non-standard pinout – most DB-9 serial hardware you'll find uses the IBM-PC standard. Use the VAXstation 2000s instructions below, and be suspicious of random DB9 hardware you find.

The VAXstation 3100 has two sort-of phone-like connectors coming out the back (arrows icon: '`/dev/ttys0`' and printer icon: '`/dev/ttys1`'. On athena, '`/dev/ttys0`' doesn't seem to be usable). The connector is very similar to the phone connector that are used on the robots, except that the springy tab is offset. When you find the right male connector, you'll need to make a phone cable. Because the VAX uses opposite pinout of our RJ11 standard, you'll need to make the cable with the wires STRAIGHT THROUGH; the colors should not swap sides, unlike the normal phone cables used elsewhere. But don't worry, you can't confuse a VAX cable with a normal cable because the special VAX connector can't be plugged in to standard RJ11 jacks.

DECstations

The DECstation 3100 has two sort-of phone-like connectors coming out the back (arrows icon: '`/dev/tty00`' and printer icon: '`/dev/tty01`'). See VAXstation 3100 below for how to wire up to these connectors.

DECstation 5000 has two IBM-standard male DB-25's coming out of the back of the box ('`//dev/tty00`' and '`/dev/tty01`'), and you can use the standard instructions below.

B.2 Serial adapter construction

****

WARNING WARNING WARNING

****

Different brand phone adapters use different color schemes!! Read the following paragraph to find out which type of phone adapter you have. (Depending on which kind you have, you may need to swap "Yellow" and "Black" in the following instructions).

Some phone adapters have 4 wires, some have 6. For those that have 6, ignore the outside two (the blue and the white). You can cut them off if you want.

Your adapter should have red and green wires in the middle, and black and yellow wires towards outside of these two. But different adapters swap the order of the red/green and black/yellow pairs!

Pick up the phone plug portion of the phone adapter you need for your machine. Orient it so that the individual colored wires are coming out towards you, and so that they are coming from the lower half. You should see that the black wire is on the left and the yellow wire is on the right. IF THIS IS NOT THE CASE (it isn't for some brands of adapters), PLEASE REVERSE BLACK AND YELLOW IN THE FOLLOWING INSTRUCTIONS.

If you have to make a phone cable, be sure to wire your phone cables like the phone company does, which means ALL WIRES ARE SWAPPED. There is typically a ridge on one of the flat faces of the cable. If you put the ridge in the same direction with respect to each plug, the wires will get swapped.

Phone to female DB25 adapter (IBM, Mac "Modem",DECstation 5000):

Yellow: pin 2

Black: pin 3

Red+Green: pin 7

Phone to male DB25 adapter (SPARCstation 1):

Yellow: pin 2

Black: pin 3

Red+Green: pin 7

Phone to female DB25 adapter (Mac "Printer" cable):

Black: pin 2

Yellow: pin 3

Red+Green: Pin 7

Phone to female DB9 adapter (IBM PC and compatibles):

Black: pin 2

Yellow: pin 3

Red+Green: pin 5

Phone to female DB9 adapter (VAXstation 2000):

Yellow: pin 2

Black: pin 3

Red+Green: pin 7

B.3 Ordering the adapters

The specification of male or female relates to the gender of the adapter itself. For example, if you have a SPARCstation with a female DB25 on the back, you would order 046-0008-ND, the male DB25 to RJ11 adapter.

If you want to order parts from Digikey to build the adapter, you can call 1-800-DIGI-KEY and give them the appropriate part number. We aren't associated with Digikey in any way, but they are one of the few vendors that will deal with small parts orders, so I include their part numbers here.

```
046-0002-ND     Male DB9-RJ11 adapter
046-0003-ND     Female DB9-RJ11 adapter
046-0008-ND     Male DB25-RJ11 adapter
046-0009-ND     Female DB25-RJ11 adapter
```

The RJ11 cable is just the same as you use to hook your phone to the wall and can be purchased much more cheaply at your local hardware store than from Digikey.

Table of Contents