DSPLinux™ TMS320VC5471 Board Support Package

User Guide

Version 1.1

205 North 10th Street Fourth Floor
Boise, ID 83702
Telephone 208.331.2226 • Fax 208.331.2227
www.ridgerun.com and www.dsplinux.net

Copyright notice

Trademarks

Product number

5471-UG0011-20020420

CONTENTS

CHAPTER 1 Introduction to the DSPLinux™ TMS320VC5471 Board Support Package

The DSPLinux TMS320VC5471 Board Support Package (BSP) specifically targets the TI™ C5471 chip. The software in this package provides support for developers who are creating applications that utilize the multi-core architecture, featuring an integrated on-chip Ethernet 10/100 MAC, found in the Texas Instruments™ TMS320VC5471 SoC.

Note: For simplification, C5471 is used in place of TMS320VC5471 in the remainder of this document

Benefits

The DSPLinux C5471 BSP is the industry's first complete operating system development package available for developers creating applications for the TI C5471 chip.

With the DSPLinux BSP, you get a complete out-of-the-box Linux® development package, containing the DSPLinux operating system, drivers, utilities, a simulator, and a complete ARM + DSP code generation toolchain that enables developers to more quickly create and debug applications for new products, which incorporate TI's general-purpose C5471 processor.

Contents

The DSPLinux C5471 BSP includes the DSPLinux TI TMS320VC5471 EVM BSP compact disk (CD-ROM). The CD-ROM contains the tools listed below to help developers get a jump-start on developing applications.

- GNU ARM7® tool chain
- DSPLinux kernel for C5471
- GDB debuggers
- Drivers for C5471 peripherals
- File system with basic system utilities
- DSPLinux Appliance Simulator for C5471
- GNU x86 tool chain

Readme

For quick installation instructions and updates not found in this document, see the README file located at the top level of the DSPLinux C5471 BSP CD-ROM.

CHAPTER 2 Installing the DSPLinux C5471 BSP

In this chapter, you will learn what host system is needed to support the DSPLinux C5471 BSP, how to obtain updates, how to install and uninstall the software, and how to configure the DSPLinux C5471 evaluation module (EVM).

SYSTEM REQUIREMENTS

Listed below are the minimum requirements for installing and using the DSPLinux C5471 BSP:

- A host computer with an Intel® Pentium® III processor, 600 MHz or faster
- A display size of 1024 x 768 or larger, bit depth of 16 bpp
- 128 MB RAM
- 750 MB free hard disk space
- Red Hat® Linux 7.1 (complete install) configured on the host computer
- X Window System with GNU Networked Object Model Environment (GNOME) or the K Desktop Environment (KDE) window manager
- A network card installed and configured on the host computer
- A Dynamic Host Configuration Protocol (DHCP) server on your network, or an available static Internet Protocol (IP) address
- TI C5471 EVM with power supply
- Serial cable (male to female, 9-pin, straight through)

Download updates

New versions of the DSPLinux C5471 BSP are released periodically. Visit the “Members Section” of http://www.dsplinux.net to check for updated software or documentation.

Note: Be sure that you have the latest release before installing the software.

DSPLINUX C5471 BSP SOFTWARE

Software installation

Follow the steps below when installing Linux and the DSPLinux C5471 BSP software; these steps also include configuring the license server and user environment.

1. Install Linux on the host computer

Install and configure Red Hat Linux 7.1 on your host computer. When choosing the type of installation, select **CUSTOM**, then scroll to the bottom of the package list and check the box marked **EVERYTHING**.

You can download or purchase Red Hat Linux 7.1 from http://www.redhat.com.

Note: Before proceeding, ensure that you can perform the following tasks on your host computer:

i) Boot into Linux.
ii) Launch the graphical user interface.
iii) Bring up a terminal window.
iv) Login as a *normal* user (not "root")

2. Install the DSPLinux C5471 BSP

You can install the DSPLinux C5471 BSP from the CD-ROM provided by RidgeRun.

– or –

If you downloaded the ISO9660 image, you will need to **mount** it as follows. (The path in the mount command below, assumes that you saved the file to "`/tmp`"; if you saved it to another directory, replace "tmp" with the appropriate directory.)

```
$ mkdir /tmp/bspmnt
$ su
# mount -o ro,loop /tmp/DSPLinux-c5471-BSP.iso /tmp/bspmnt
# exit
```

After completing the "mount" command above, the installation is essentially the same whether you have a CD-ROM or the downloaded ISO9660 image.

a. Go to the installation directory.

```
$ cd /tmp/bspmnt
```

– or –

```
$ cd /mnt/cdrom
```

b. Run the installation script specifying the development directory. Be prepared to enter the “root” password at this point. The “root” password is necessary when writing to certain directories.

```
$ ./INSTALL <development directory>
```

The DSPLinux C5471 is now installed and ready for configuration.

Note: The *<development directory>* is the parent directory for the kernel source and all other user level files. This should typically reside in a location to which the user has write access. An example of a safe location is your “home” directory as shown below:

```
./INSTALL /home/smith/projects
```

Note: If, in your environment, you share the GNU toolchain (provided by DSPLinux) and other tools on a network server, you can install the DSPLinux BSP without installing the toolchain.

To omit the tools, located in `/opt/DSPLinux`, enter the command below.

```
$ ./INSTALL --notools <development directory>
```

File locations

- The source code, object files, makefiles, and documentation are in: `<development directory>/dsplinux/c5471`
- The toolchain for building DSPLinux applications is in: `/opt/DSPLinux`

Uninstalling the software

To remove a previously installed version of the DSPLinux C5471 BSP (all files in `/opt/DSPLinux/`), perform the actions below as **ROOT**.

CAUTION: Before removing the tree, be sure to backup any files you want to keep.

```
$ su -c ‘rpm -e DSPLinux-TIDSC21_EVM-pgui’
$ su -c ‘rpm -e DSPLinux-X86SIMu-pgui’
$ su -c ‘rpm -e DSPLinux-TIDSC21_EVM-tools’
$ su -c ‘rpm -e DSPLinux-X86SIMu-tools’

## used by this and other DSPLinux tools
$ su -c ‘rm -rf /opt/DSPLinux/rules’
```

To remove the DSPLinux source code, objects, documentation, and makefiles; simply remove the *<development directory>*`/dsplinux` tree, using the following command.

```
$ rm -rf <development directory>/dsplinux/c5471
```

TURNING ON THE C5471 EVM

Follow the steps below to configure DSPLinux on the C5471 EVM.

1. Observe proper static discharge precautions.
2. Make sure that the C5471 EVM is turned off.
3. Configure the EVM jumpers. For details, see Appendix A, "Jumpers for the C5471 EVM" on page 127.
4. Connect the serial port to your host using a male to female, 9-pin, straight through serial cable.
5. Launch minicom or an equivalent serial port terminal program on the host using the following settings:

   115200 baud, no parity, 1 stop bit, 8 data bits (8N1)

6. Apply power to the EVM. If the board does not boot, press **RESET** to start the board.
7. The DSPLinux kernel will send a welcome message to the terminal display as shown below.

```
Welcome to the DSPLinux ARM7 distro by RidgeRun, Inc.

Copyright (C) 2001 RidgeRun, Inc.
Please report all bugs and problems to <support@dsplinux.net>
```

8. To log into the EVM, enter the user name "root" and press **ENTER**.
   The message below displays on the screen

```
rr-c5471 login:

Welcome to the DSPLinux(TM) by RidgeRun, Inc.

BusyBox v0.61.pre (<date and time>) Built-in shell (msh)
Enter 'help' for a list of built-in commands.
#
```

The C5471 EVM is now ready for use.

CHAPTER 3 Using the C5471 Target Hardware

This chapter provides the necessary information for working with DSPLinux on the C5471 target, including the steps used to re-configure the kernel or add a new program to the root file system. In addition, we cover the necessary steps for re-building a new system and for getting the new system on the target hardware using the rrload bootloader.

You can build the kernel and the root file system individually on an as needed basis. In the sections that follow, our examples build only the portion of the system actually being changed; although it should be noted that a top-level build can always be performed, which will build the entire system including applications, file system, kernel, and bootloader. Generally this is more time consuming than a more focused build is, but if you are in doubt as to what needs to be rebuilt, we recommend a top-level build.

A **top-level** build can be performed with the following commands:

```
...Top-Level system build...

$ cd dsplinux/c5471
$ . ./setenv
$ make
```

Or, instead, you can use the following commands for focused **individual** builds:

1. To re-build a particular set of apps:

   ```
   $ cd dsplinux/c5471
   $ . ./setenv
   $ cd dsplinux/c5471/apps/hello
   $ make all install
   $ cd dsplinux/c5471/apps/leds
   $ make all install
   ```

   To get a new file system containing the newly built/installed apps, re-build the root file system too. See below.

2. To re-build the root file system:

   ```
   $ cd dsplinux/c5471
   $ . ./setenv
   $ make fs
   ```

3. To re-build the kernel:

```
$ cd dsplinux/c5471
$ . ./setenv
$ make linux
```

4. To re-build the bootloader:

```
$ cd dsplinux/c5471
$ . ./setenv
$ make rrload
```

Of course, you will have no reason for re-building any of these components unless you are interested in changing the as-shipped BSP system. Those changes, if any, will likely fall into the following two categories.

- Changing the kernel configuration to turn on or off various kernel features (see “Changing the kernel configuration” on page 10).
- Changing the root file system to contain a different set of target applications or Linux Modules (see “Changing the Root file system” on page 12).

These forms of changes are discussed in the sections that follow.

CHANGING THE KERNEL CONFIGURATION

The DSPLinux kernel supplied with your BSP is pre-configured and immediately ready for use. However, if your objectives require components that are not contained in the default DSPLinux kernel, you may want to change the kernel configuration. In the traditional manner, the kernel configuration is expressed in the “.config” [working copy] and “config” [master backup copy] files at the locations shown below.

working copy.	`dsplinux/c5471/linux/.config`
master backup copy	`dsplinux/c5471/build/config`

Changing the DSPLinux kernel configuration requires changing the “.config” working copy. We recommend that it **NOT** be modified directly, but rather that you modify it using a standard method as shown in the steps below where we use **make xconfig** to modify “.config”.

1. Type the commands below to change the kernel configuration.

```
$ cd dsplinux/c5471/linux
$ make xconfig
```

2. The **xconfig** session displays the kernel options and allow you to *change* them. As a last step in making those changes, be sure to **SAVE** and **EXIT** before proceeding.

3. With the **xconfig** session complete, you can build the new DSPLinux kernel as follows:

```
$ cd dsplinux/c5471/linux
$ . ../setenv
$ make clean dep
$ make linux
```

Note: Doing the “make clean dep” step is always recommended when making changes to the DSPLinux configuration. However, it is not necessary with each subsequent rebuild which might occur as part of an iterative I/O driver development process, for example.

The resulting kernel image is expressed in the two ways below.

A. `dsplinux/c5471/linux/linux`

An **ELF** object format for use with JTAG based target debug tools.

B. `dsplinux/c5471/linux/linux.rr`

An **rrbin** format for use when downloading to the target using the rrload bootloader.

RESETTING THE DSPLINUX KERNEL TO FACTORY DEFAULTS

Should you want to abandon your re-configuration efforts and return to the factory supplied default settings, you can accomplish this with the steps below.

```
$ cd dsplinux/c5471/linux
$ make clean
$ make mrproper
$ cp ../build/config .config
$ make oldconfig
$ make dep
$ make
```

CHANGING THE ROOT FILE SYSTEM

If desired, you can configure the “romfs” file system with additional content not included in the EVM’s default file system. A typical example would be adding a new program that can then be invoked on some future run of the C5471.

In the following steps, we show two methods you can use to put the “hello” application into the file system image: the Makefile method and the Manual method. These steps are performed on the desktop prior to using the new file system with the EVM.

Makefile method

1. Using the following commands the “hello” example application is compiled. Then the compiled program is installed into the file system, using **Makefile**. The “hello” Makefile is very simple and primarily designed to be used as a template for new DSPLinux applications.

   ```
   $ cd dsplinux/c5471/apps/hello
   $ . ../../setenv
   $ make all install
   ```

2. The final step is to build the file system “romfs” image, which will be downloaded to the board using rrload.

   ```
   $ cd dsplinux/c5471
   $ . ./setenv
   $ make fs
   ```

3. The **make fs** command creates the romfs image as “dsplinux/c5471/linux/romdisk.rr”, which can be downloaded to the board using rrload (See Chapter 11 “The DSPLinux Bootloader (rrload)” on page 81).

Manual Method

If you examine the “hello” Makefile used above, you will see the DSPLINUX_FS variable. This variable will generally switch between two values, depending on which **setenv** script you source (hardware or simulator).

- For a hardware target the value is: `dsplinux/c5471/fs/TIDSC21_EVM/`
- For simulator targets the value is: `dsplinux/c5471/simulator/fs/X86SIMu/`

  The directory structures appear as follows:

  Hardware

  ```
  dsplinux/c5471/fs/TIDSC21_EVM
  |-- binaries
  `-- scripts
  ```

  Simulator

  ```
  dsplinux/c5471/simulator/fs/X86SIMu
  |-- binaries
  `-- scripts
  ```

1. Place the files in either the binaries or scripts directories to get them on the file system. The only difference between scripts and binaries exists in RidgeRun’s internal revision control system. You may choose to locate all files in either directory or in just one for simplicity. The last and most important note about these directories, is that they reflect the root (/) directory on the target file system.

Example

dsplinux/c5471/fs/TIDSC21_EVM/binaries/opt/mypackage/bin/myprogram will show up in the target file system as **/opt/mypackage/bin/myprogram**.

2. Just as in any file system build method, it is necessary to rebuild and install the new image to the board.

   ```
   $ cd dsplinux/c5471
   $ . ./setenv
   $ make fs
   ```

3. The above commands create `dsplinux/c5471/linux/romdisk.rr`, which can be downloaded to the board using rrload (See Chapter 11 “The DSPLinux Bootloader (rrload)” on page 81).

PUTTING THE NEW SYSTEM ON THE TARGET

If you have constructed a new kernel and/or file system, you may test it without altering the EVM board's currently flashed system. This involves using the EVM board's bootloader (rrload) to download either a new kernel or a new file system or both. Once the components are brought into RAM memory, you can instruct the bootloader to transfer control to the kernel and the system will be booted with your new system. However, this new system is **lost** on the next power cycle.

If you have a system with which you want to replace the currently flashed system, you may repeat the process and add the optional bootloader steps to erase and re-flash the on-board kernel and/or file system (see step 4 and step 5 below).

rrload bootloader

The **rrload** instructions are described in more detail in Chapter 11 "The DSPLinux Bootloader (rrload)" on page 81.

The steps below describe the general procedure for testing the new kernel, file system, or both.

1. Power cycle the EVM while holding down the **ENTER** key for a few seconds. This intercepts the normal boot process and ensures that the EVM board's bootloader user interface (UI) is presented to the user over the serial port.

2. From the bootloader's main menu, select **OPTION 3** to display the rrload **Params** window. Verify that the settings are appropriate for the operation(s) that you will perform and make any necessary changes.

3. Use the **rrload UI** to download the new kernel and/or file system.

Note: Steps 4 and 5 below are optional steps. Use them only when you wish to erase or store the new kernel or file system

4. Erase the on-board kernel and/or file system (*optional*).

   If you have a system with which you want to replace the currently flashed system, use the *optional* rrload UI to **ERASE** the original flashed kernel and/or file system (for details, see Chapter 11 "The DSPLinux Bootloader (rrload)" on page 81).

5. Store the on-board kernel and/or file system (*optional*).

   To store the new kernel and or file system into flash, use the rrload UI **Store** command (see Chapter 11 “Kernels and file systems can be stored two ways...” on page 92).

6. Issue the **boot_auto** command. This command is more elaborate than the simple **boot** command and ensures that both a file system and kernel exist in RAM prior to transferring control to the kernel.

The boot_auto command and what it does...

- If you download *only* a “new kernel,” the **boot_auto** command attempts to locate a file system in flash, move it to RAM, and combine it with your new kernel to form the full system.
- Similarly, if you download *only* a “new file system,” the **boot_auto** command attempts to locate a kernel in flash and move it to RAM, forming the final system before finally transferring control to the kernel.

An example of the command is: `rrload> boot_auto`

RUNNING PROGRAMS THAT WRITE TO THE FILE SYSTEM

In the default usage scenario, the kernel mounts a “romfs” file system. The entire file system is read-only. The only exception is the **ramdisk** created by the system startup scripts. One ramdisk is mounted at mnt/ramdisk. The following `directories: /root, /tmp`, and `/var are links to /mnt/ramdisk`. These directories are read-write; however, at the next power cycle, you will **lose** any content placed there while running the C5471.

If you are interested in writing to other parts of the file system at runtime (with contents surviving device power cycles), we suggest that you switch to a **NFS mount** of the root file system. For details on this process, see the Chapter 5 “Enabling Device Networking” on page 35.

CHAPTER 4 Using the GDB Debuggers

THE PURPOSE OF A GDB DEBUGGER

The purpose of a debugger such as GDB is to allow you to see what is going on inside another program while it executes. Typically, the term “program” refers to a binary image residing in the target’s file system and, when invoked, is loaded by the DSPLinux operating system. Three debuggers are provided as described below.

- The first is **arm-uclinux-gdb**, which is a GDB that facilitates the form of debugging just described.

  Additionally, the DSPLinux BSP includes two other GDB debuggers for use with more specialized logic debugging.

- One is **sdi-arm-gdb**, which is used when debugging any kernel driver linked with the DSPLinux kernel, or for debugging the DSPLinux kernel itself, or the board's bootloader, and so forth. It is used for images running on the GPP processor.
- The other GDB debugger is **sdi-dsp-gdb**, which is used for debugging logic running on the DSP processor.

The last two mentioned are specialized GDB debuggers that work with the *Spectrum Digital GDB server* which is running, **NOT** on the target EVM, but rather on a network connected Window’s personal computer (PC); the Window’s PC is connected to the target EVM via a parallel cable. The Spectrum Digital GDB server uses the parallel cable to interface to the board’s parallel JTAG port. If the board offers only a traditional JTAG connector, instead of a parallel port JTAG, then the PC will instead connect to a parallel-to-JTAG adaptor pod such as the *Spectrum Digital model XDS510PP*, which then connects to the board’s standard JTAG connector.

All three debuggers are based on the standard GNU GDB debugger and offer the same command line user interface. They are provided within the tool chain’s “bin” directory of your DSPLinux BSP installation.

For information regarding the GDB debugger UI, please refer to one of the following:

- The standard **gdb** *man pages*.
- The *info pages* included on your Linux desktop.
- The online manual provided by the *Free Software Foundation* at http://www.gnu.org/manual/gdb-4.17/gdb.html.

Three debuggers provided by the DSPLinux BSP

```
/opt/DSPLinux/TIDSC21_EVM/crossdev/bin/arm-uclinux-gdb
/opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-arm-gdb
/opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-dsp-gdb
```

Note: Each of these reads the optional ~/.gdbinit on startup.

Using a graphical front end

If you are interested in using a *graphical front end* with the supplied GDB debuggers, there are several industry available front ends designed for GDB. One example is the free “ddd” utility. You can install **ddd** on your Linux desktop from the following site: http://www.gnu.org/software/ddd/

Below is an example of how you might invoke one of the DSPLinux GDB debuggers, using a graphic front end.

Example 1: Invoke a DSPLinux debugger

On the Linux desktop, type one of the following:

```
$ ddd --debugger arm-uclinux-gdb
```

–or–

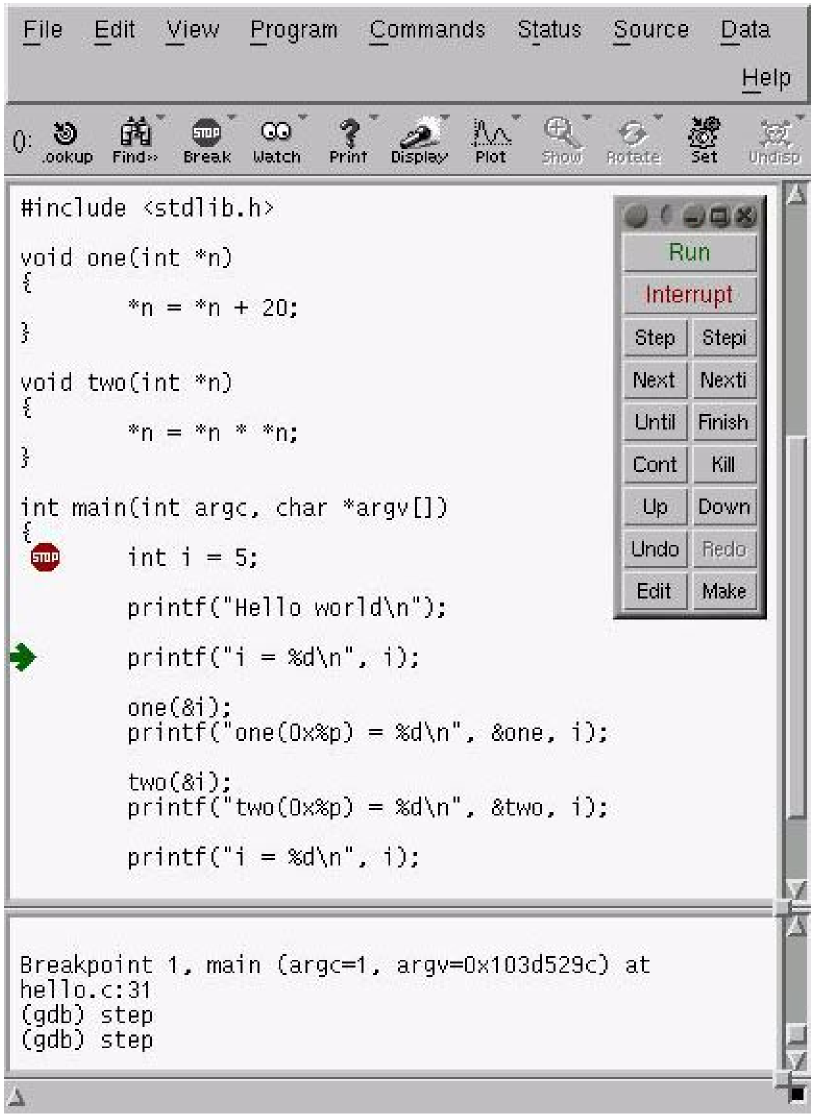
```
$ ddd --debugger sdi-arm-gdb
```

–or–

```
$ ddd --debugger sdi-dsp-gdb
```

When using **ddd**, a typical session might look like the figure below.

Figure 1. Typical session using “ddd”



```
#include <stdlib.h>

void one(int *n)
{
        *n = *n + 20;
}

void two(int *n)
{
        *n = *n * *n;
}

int main(int argc, char *argv[])
{
        int i = 5;

        printf("Hello world\n");

        printf("i = %d\n", i);

        one(&i);
        printf("one(0x%p) = %d\n", &one, i);

        two(&i);
        printf("two(0x%p) = %d\n", &two, i);

        printf("i = %d\n", i);
```

```
Breakpoint 1, main (argc=1, argv=0x103d529c) at
hello.c:31
(gdb) step
(gdb) step
```

DEBUGGERS CONCEPTUALLY DIAGRAMMED

See the figures below for 3-debugging scenarios.

Figure 2. Debugging a program running on Linux

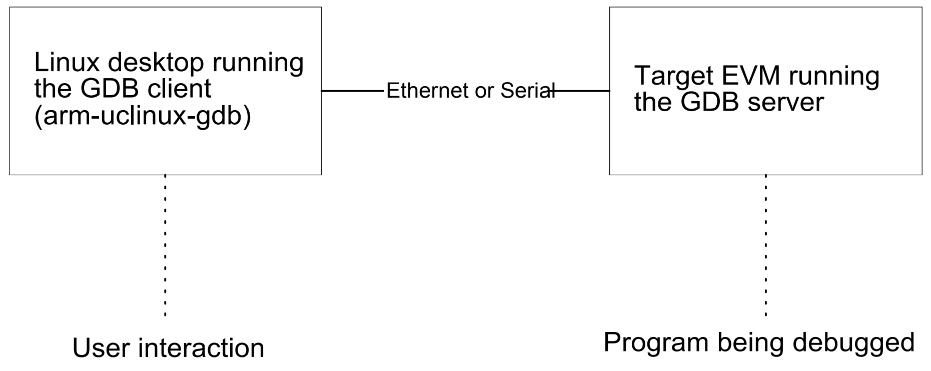

Linux desktop running the GDB client (arm-uclinux-gdb)
Ethernet or Serial
Target EVM running the GDB server
User interaction
Program being debugged

Figure 3. Debugging a Linux device driver, Linux kernel, the bootloader, or such

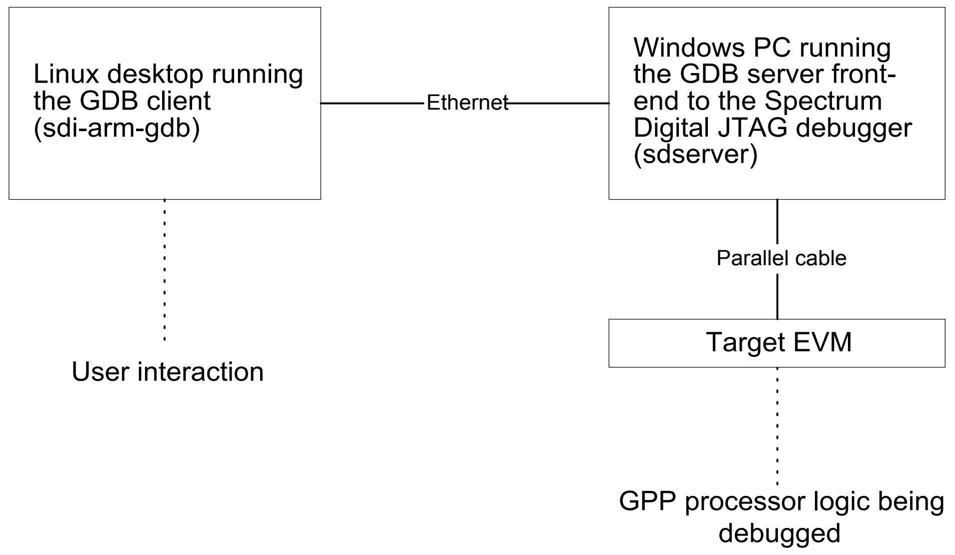
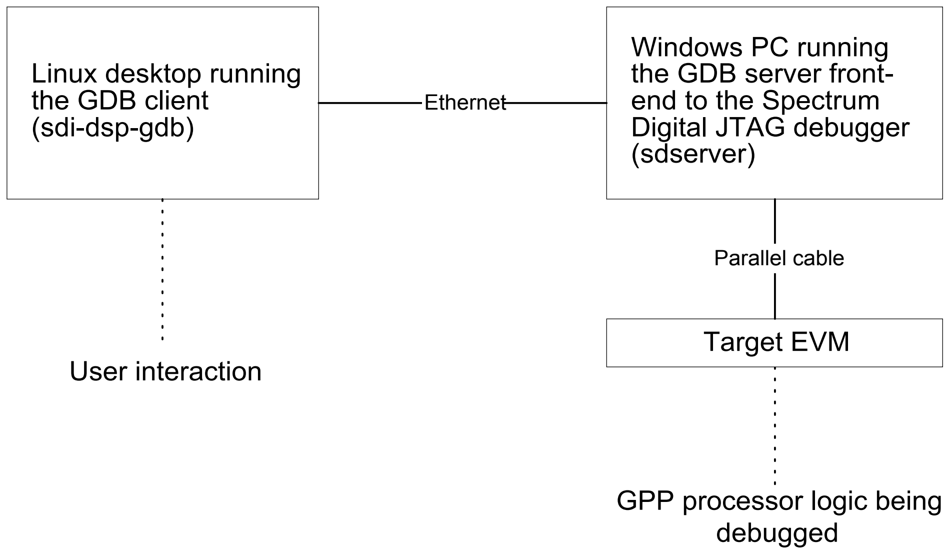

Linux desktop running the GDB client (sdi-arm-gdb)
Ethernet
Windows PC running the GDB server front-end to the Spectrum Digital JTAG debugger (sdserver)
Parallel cable
Target EVM
User interaction
GPP processor logic being debugged

Figure 4. Debugging a DSP program


Linux desktop running the GDB client (sdi-dsp-gdb)
Ethernet
Windows PC running the GDB server front-end to the Spectrum Digital JTAG debugger (sdserver)
Parallel cable
Target EVM
User interaction
GPP processor logic being debugged

GPP PROGRAM DEBUGGER (ARM-UCLINUX-GDB)

You can use the **arm-uclinux-gdb/gdbserver** pair to debug traditional Linux programs on the remote target. The first of the pair runs on the Linux desktop and the second of the pair runs on the target with the program you are debugging.

When developing applications that run on the DSPLinux operating system, you will use this debugger. You can use any program invoked from the DSPLinux command line with this debugger, providing the program was previously configured to be compatible with GDB. To help you in this area, there is a “apps/hello” program included in the BSP installation that shows a Makefile containing details necessary to prepare an application for debugging. Specifically, you should be aware of the two issues concerning...

- The “flat” format file referenced by the target side GDB, and
- The “gdb” format file referenced by the desktop side of GDB.

Preparing and debugging the program

1. On the desktop, you will need the "gdb" version of your program (the specially prepared `*.elf` file that goes by the name `*.gdb`) in the same directory where you will invoke **arm-uclinux-gdb**.

2. On the GPP target machine, invoke the **gdbserver** in the directory where you have the file you are debugging; the "flat" format version of your program.

3. Be sure to place the debuggable version of your program into the target's file system prior to starting the debugger. You may do this by...

   a. Using the *BSP crossdev environment* to build a new file system that the target then either mounts via NFS or you can download the new file system via rrload on the target prior to booting the kernel.

   –or–

   b. **FTP example:**
   You may use FTP from the desktop to "put" your program onto the file system of the Ethernet connected running target.

   –or–

   c. **Kermit example:**
   From the target, use **gkermit** to "receive" the file from the desktop, for example, over the serial line.

   Once you get your program placed in the target's file system, it is ready for debugging.

Note: Check the file permissions to insure that the program has execute privileges. If not, you can add it with the command below.

```
# chmod +x [file]
```

4. For debugging there are two methods for communication between the target's **gdbserver** and the desktop **GDB client**. One uses an available Serial port while the other uses the Ethernet port.

   Some target EVM boards have multiple serial connectors and therefore can assign one of them to the system console (`/dev/ttyS0`) while the other (`/dev/ttyS1`) is assigned for GDB communication with the desktop. In the case where the target has only one serial port, it is often dedicated to the system console and therefore the Ethernet port can be used for GDB communication with the desktop.

If you are going to use the serial port for debugging, see Chapter 9 “The Serial Port” on page 71 for additional information pertaining to the use and configuration of the board’s serial port. Use the connection that is appropriate for your target hardware, specifying it at the time you start the debug session. In addition, the command line indicates the name of the program you intend to debug; the “flat” binary Linux program format intended to run on the target.

The steps in this example debug the *hello* application, which was built by typing **make** within the BSP’s `apps/hello` directory.

5. To start the target-side of the GDB session, do one of the following:

   a. If using a **network** GDB connection, enter the text below.

```
# gdbserver :5471 hello
```

Note: The port 5471 was selected randomly as a high port number. Any high port number that doesn't conflict with existing active ports will work.

–or–

   b. If using a **serial** GDB connection, enter the text below.

```
# gdbserver /dev/ttyS1 hello
```

Note: For the C5471 EVM, we recommend an Ethernet GDB connection since `/dev/ttyS0` is dedicated to the system console.

6. Start the desktop-side of the GDB session by invoking the **arm-uclinux-gdb** debugger. This establishes a connection to the server previously started on the remote target.

   We invoke **arm-uclinux-gdb** while in the directory containing the “gdb” version of our program file (`hello.gdb`). Then issue commands at the GDB command line prompt to establish the connection and load the program’s debug symbols.

Note: In next step below, change the commands based on the following:

(1) Use the PC-address reported by the prior GDB “target” command.

(2) Replace the x.x.x.x string with the actual IP of the EVM device. If you are not sure what the IP is, type “ifconfig” on the target command line to generate a network configuration report.

(3) The port 5471 was selected randomly as a high port number. Any high port number that doesn't conflict with existing active ports will work.

(4) If using “ddd” graphical front end, invoke using the following command.

$ ddd --debugger /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/arm-uclinux-gdb

a. If using a **network** GDB connection (GDB client [Linux Desktop]), enter the text below.

```
$ cd dsplinux/c5471/apps/hello

$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/arm-uclinux-gdb

(gdb) set endian little

(gdb) target remote x.x.x.x:5471

(gdb) add-symbol-file hello.gdb PC-address
```

–or–

Note: In item 6.b below, change the commands based on the following:

(1) Use the PC-address reported by the prior GDB “target” command.

(2) If using “ddd” graphical front end, invoke using the following command.

$ ddd --debugger /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/arm-uclinux-gdb

b. If using a **serial** GDB connection, enter the text below.

```
$ cd dsplinux/c5471/apps/hello
$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/arm-uclinux-gdb
(gdb) set endian little
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS1
(gdb) add-symbol-file hello.gdb PC-address
```

7. At this point you can use the normal GDB session commands to set breakpoints, run, single step, examine variables, and so forth.

The Simulation environment

Debugging in the simulation environment is very similar. You will want to source the **setenv.sim** file prior to building your executable. Sticking with the *apps/hello* example, only the executable (“hello”) is created.

1. The executable should be placed on the file system of the simulated target. Because of a simulator serial connection issue, you will use the network protocol below to connect in the simulator.

```
$ gdbserver :5471 hello
```

2. On the debugging host and while in the same directory containing the executable to debug, type the command below to invoke GDB.

```
$ gdb
```

3. Type the command below to add the symbol file.

```
(gdb) file hello
```

Note: In step 4 below, make the appropriate replacements in the command based on this note:

(1) Replace the x.x.x.x string with the actual IP of the EVM device. If you are not sure what the IP is, type `ifconfig` on the target command line for a network configuration report.

(2) The port 5471 was selected randomly as a high port number. Any high port number that doesn't conflict with existing active ports will work.

4. Use the following command to connect to the remote host.

```
(gdb) target remote x.x.x.x:5471
```

5. You are now ready to debug in the simulated environment.

More about arm-uclinux-gdb

Important notes

Because **/dev/ttyS0** is the default console, applications should not use it for other purposes. Instead, if it is available, the application(s) should use **dev/ttyS1** or use the **Ethernet port**.

By default, the target serial ports start up at 115200 baud, whereas most host PC's start up at 9600 baud. This *must remain compatible* with the *bootloader*, which uses **115200** to maximize download speed. A single minicom session can communicate with the bootloader and then with the booted kernel without changing any settings. Applications that use a different baud rate can use the standard utilities or **ioctl** calls for *changing serial port speed*.

Bugs

The serial connection method for the simulator does **NOT** work. The work-around is to use the network connection in the simulation environment.

To use the network connection in the simulator, the user must run **/usr/bin/um_eth_net_route** on the target (simulator) to enable a network connection to the host from which it was launched.

See also

- gdb manpage
- apps/hello/hello.c
- apps/hello/Makefile

GPP rrload/kernel debugger (sdi-arm-gdb)

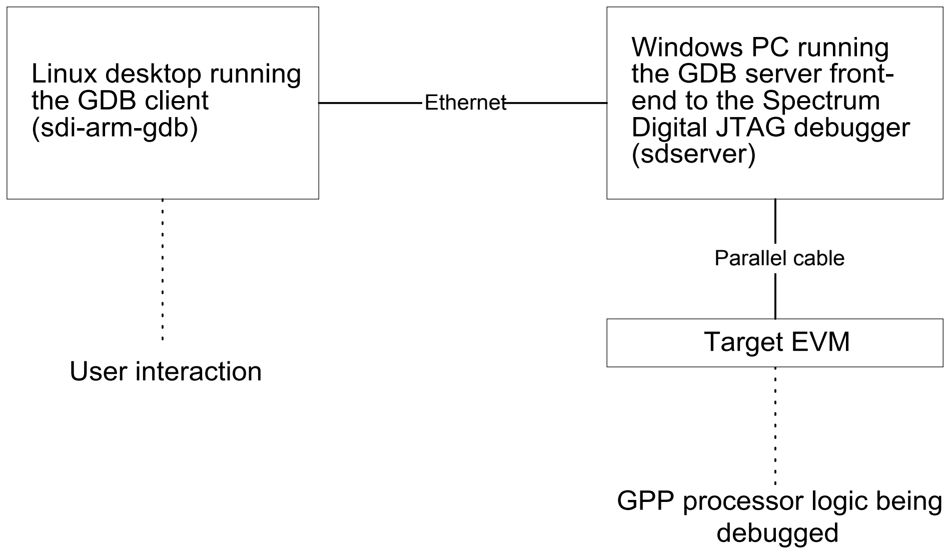
Note: The sdi-arm-gdb GDB Linux client is supplied with your DSPLinux BSP, however, the GDB Windows PC server is supplied by Spectrum Digital for use with their JTAG based debugger.

The **sdi-arm-gdb** GDB debugger provided with the DSPLinux BSP is intended for debugging *non-traditional GPP programs*. In other words, binary images that are not loaded and run by the DSPLinux operating system (OS).

For example, the **rrload bootloader** is an *ELF* image that runs on the target's GPP, however it is not invoked by the DSPLinux kernel. In addition, the DSPLinux kernel itself is an ELF file that obviously is not invoked by the OS. These are images that are traditionally "pushed" to the board using a JTAG based debugger tool or "pulled" to the board using a board resident bootloader. This class of ELF image can be debugged using the **sdi-arm-gdb** debugger client in conjunction with a JTAG based debugger tool back-end.

In particular, the **sdi-arm-gdb** client is designed to run on the Linux Desktop and interact (via Ethernet) with the **sdserver** utility running on a Windows PC. The sdserver represents the server side (or back-end) of the client/server GDB pair. It is designed to interface (as a front-end) with the *Spectrum Digital debugger tool* running on that same PC. This debugger tool is connected to the remote target's JTAG port via a parallel cable connected to the PC. As presented earlier, the diagram in Figure 5 on page 28 summarizes these connections.

Figure 5. Linux desktop running the GDB client (sdi-arm-gdb)


Linux desktop running the GDB client (sdi-arm-gdb)
Ethernet
Windows PC running the GDB server front-end to the Spectrum Digital JTAG debugger (sdserver)
Parallel cable
Target EVM
User interaction
GPP processor logic being debugged

It should be noted that the main advantage to using the **sdi-arm-gdb** debug interface is that it allows you to conduct your debug session from the Linux desktop (instead of the Windows PC), using the familiar GDB interface.

It is **NOT** anticipated that you will need to debug the rrload bootloader or the DSPLinux kernel, rather the debugger is expected to typically have its biggest use in the two areas mentioned below.

Typical uses

- A means to install the bootloader (rrload) to a blank target.
- A means to debug device drivers linked with the DSPLinux kernel.

See the examples below (Example 2 and Example 3) for the necessary steps to get the rrload bootloader installed on the target EVM. While it is true that you most likely already have rrload installed on your board, and while it is also true that an installed rrload is capable of updating itself with newer versions of rrload without the need for JTAG, these nevertheless serve as a good examples. Moreover, these examples are ones that you may actually need one day should something catastrophic happen to your installed rrload.

Example 2: The GDB server (Windows PC)

1. Run the **sdserver** program supplied by Spectrum Digital.
2. Take note of the reported IP address.

Note: In Example 3 below, apply the appropriate information (where applicable) from this note.

(1) Replace the x.x.x.x string with the actual IP of the Windows PC running the GDB front end (sdserver) to the Spectrum Digital Debugger running on that same PC, which itself is then connected to the EVM device. This is not the IP of the remote target, rather the remote PC connected to target.

(2) The port 5471 is not random. Use this exact value.

(3) If using the 'ddd' graphical front end, instead invoke as shown below.

```
$ ddd --debugger /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-arm-gdb
```

Example 3: The GDB client (Linux desktop)

```
$ cd dsplinux/c5471/rrload
$ . ../setenv
$ make clean
$ make
$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-arm-gdb
(gdb) set iibp 1
(gdb) set endian little
(gdb) target remote x.x.x.x:5471
(gdb) #
(gdb) load setup
(gdb) symbol-file setup
(gdb) b setupdone
(gdb) cont
      When we hit the break point we have the h/w setup
      the way we need and can now load our program of
      interest into the now available SDRAM.
(gdb) # Next, remove the earlier breakpoint.
(gdb) delete 1
(gdb) load rrload
(gdb) cont
      At this point the rrload bootloader should be
      running as evident by the UI presented over
      the serial port to your remote terminal
      session (minicom, etc). If desired, use the
      rrload UI to tell this RAM based rrload to
      copy itself to on-board flash. Now rrload will
      run on subsequent target bootups. (Please see
      the rrload documentation for more information).
```

Next, in the following example, we offer a method that shows how the steps change if, instead, we want to load and/or debug the DSPLinux kernel or device driver contained within. Notice that the process *differs* **ONLY** in the *last four steps*.

Example 4: Load/debug the DSPLinux kernel or device driver

```
$ cd dsplinux/c5471/linux
$ . ../setenv
$ make
$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-arm-gdb
(gdb) set iibp 1
(gdb) set endian little
(gdb) target remote 192.168.1.209:5471
(gdb) set *0xffff2e00=(unsigned long)0xecf
(gdb) set *0xffff2e04=(unsigned long)0xec1
(gdb) set *0xffff2e08=(unsigned long)0x13db
(gdb) set *0xffff2f10=(unsigned long)0x0
(gdb) load setup
(gdb) symbol-file setup
(gdb) b setupdone
(gdb) cont
      (When breakpoint reached, proceed below.)
(gdb) delete 1
(gdb) load linux
(gdb) symbol-file linux
(gdb) b start_kernel
      This example breakpoint is a main.c "C" routine,
      which is executed very early in the DSPLinux boot
      process.
(gdb) cont
      This starts executing the kernel
      at its entry point "DSPLinux_entry"
      of the head-arm-orion.S Assembly file.
```

Example 5: Using the ~/.gdbinit GDB startup file

Recall that you can use the **~/.gdbinit** GDB startup file to encapsulate commonly issued commands and macro definitions. This allows us to simplify the debug session by placing commonly issued commands in the GDB startup file, thus eliminating the need to type them in at each session.

For example, setup the `~/.gdbinit` file as shown below.

```
$ cat ~/.gdbinit
define sdi-arm-setup
   set iibp 1
   set endian little
   target remote x.x.x.x:5471
   #
   # setup CS0 (flash)
   set *0xffff2e00=(unsigned long)0xecf
   # setup CS1 (sram)
   set *0xffff2e04=(unsigned long)0xec1
   # setup CS2 (LEDS, switches, etc).
   set *0xffff2e08=(unsigned long)0x13db
   # Arm and DSP reset control
   set *0xffff2f10=(unsigned long)0x0
   #
   load setup
   symbol-file setup
   b setupdone
   cont
end
```

This reduces the session steps to the following:

```
$ cd dsplinux/c5471/linux
$ . ../setenv
$ make
$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-arm-gdb
(gdb) sdi-arm-setup
      (When breakpoint reached, proceed below.)
(gdb) delete 1
(gdb) load linux
(gdb) symbol-file linux
(gdb) b SomeSymbolOfInterest
(gdb) cont
```

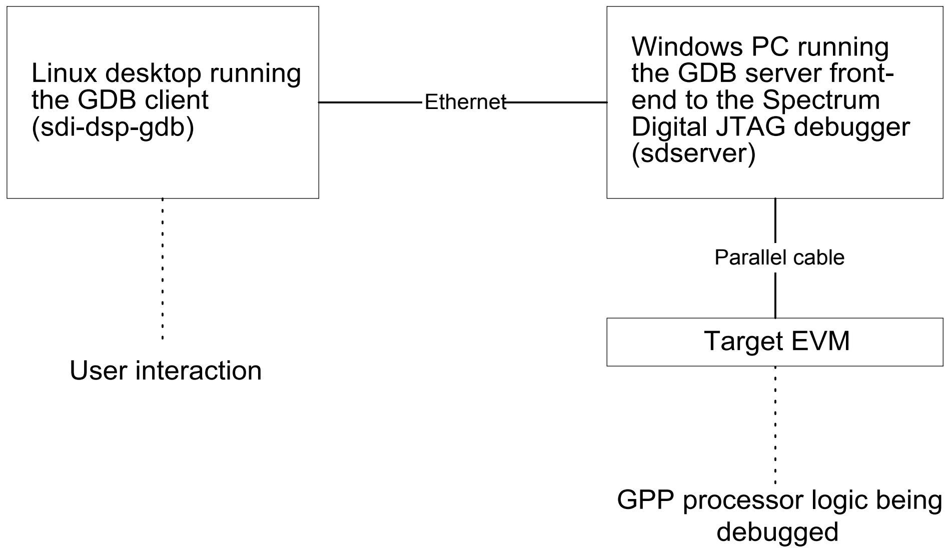
DSP PROGRAM DEBUGGER (SDI-DSP-GDB)

Note: The sdi-dsp-gdb GDB Linux client is supplied with the DSPLinux BSP, however, the GDB Windows PC server is supplied by Spectrum Digital for use with their JTAG based debugger.

The **sdi-dsp-gdb** GDB debugger provided with the DSPLinux BSP is intended for debugging DSP programs.

In particular, the **sdi-dsp-gdb** client is designed to run on the Linux desktop and interact (via Ethernet) with the **sdserver** utility running on a Windows PC. The **sdserver** represents the server side (or back-end) of the client/server GDB pair. It is designed to interface (as a front-end) with the *Spectrum Digital debugger tool* running on that same PC. This debugger tool is connected to the remote target's JTAG port via a parallel cable connected to the PC. As presented earlier, the diagram below illustrates these connections.

Figure 6. Linux desktop running the GDB client (sdi-dsp-gdb)


Linux desktop running the GDB client (sdi-dsp-gdb)
Ethernet
Windows PC running the GDB server front-end to the Spectrum Digital JTAG debugger (sdserver)
Parallel cable
Target EVM
User interaction
GPP processor logic being debugged

The example below presents the steps necessary to debug a DSP program. In this case, for illustrative purposes, we are debugging the **blink_c_dsp** program supplied with the BSP.

Example 6: The GDB server (Windows PC)

1. Run the **sdserver** program supplied by Spectrum Digital.
2. Make a note of the reported IP address.

Note: In below, modify the

(1) The startup file for this GDB is ~/.tic54x-gdbinit

(2) Replace the x.x.x.x string with the actual IP of the Windows PC running the GDB front end (sdserver) to the Spectrum Digital Debugger running on that same PC, which itself is then connected to the EVM device. This is not the IP of the remote target, rather it is the remote PC connected to target.

(3) The port 5472 is not random. Use this exact value.

(4) If using the `ddd` graphical front end, invoke it using the command below.

```
$ ddd --debugger /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-dsp-gdb
```

Example 7: The GDB client (Linux desktop)

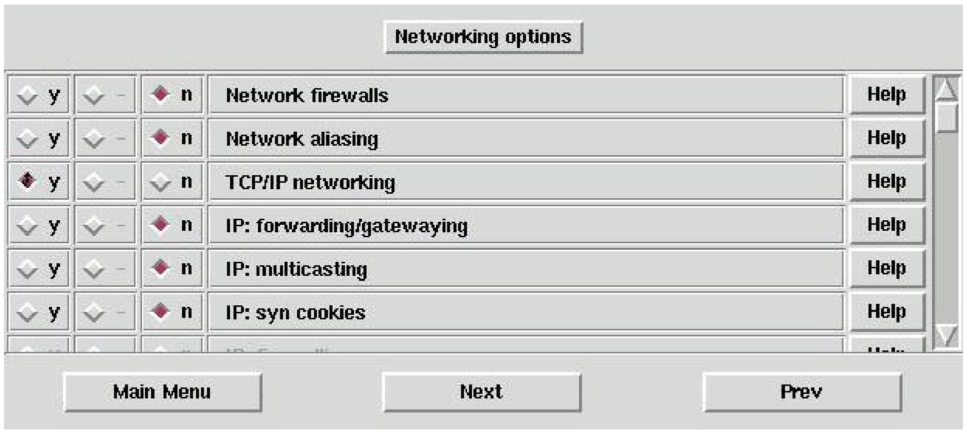
```
$ cd dsplinux/c5471/apps/blink
$ . ../setenv
$ make clean
$ make
$ /opt/DSPLinux/TIDSC21_EVM/crossdev/bin/sdi-dsp-gdb
(gdb) target remote x.x.x.x:5472 (note: must use "5472")
(gdb) set $st0=0x1800
(gdb) set $st1=0x2800
(gdb) set $pmst=0xff80
(gdb) load blink_c_dsp
(gdb) cont
```

CHAPTER 5 Enabling Device Networking

DEFAULT KERNEL CONFIGURATION WITH NETWORKING ENABLED

The DSPLinux BSP ships with a default kernel configuration that has networking enabled. Recall that kernel configuration changes are generally performed with the “make xconfig” process as run from the **dsplinux/c5471/linux** directory. A snapshot from an xconfig session with the default network settings is shown in the figure below.

Figure 7. Networking options (default settings)



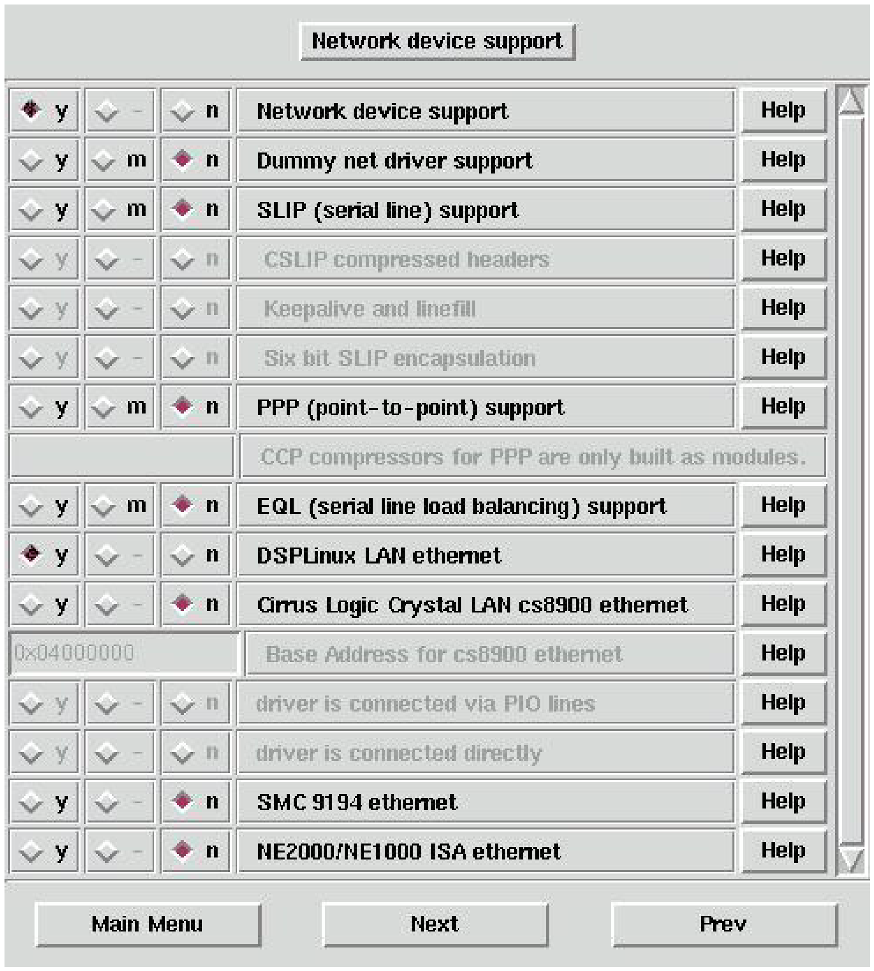
y	-	[✓ n]	Network firewalls
y	-	[✓ n]	Network aliasing
[✓ y]	-	n	TCP/IP networking
y	-	[✓ n]	IP: forwarding/gatewaying
y	-	[✓ n]	IP: multicasting
y	-	[✓ n]	IP: syn cookies

With the supplied default configuration, you will only need to use the target’s bootloader (rrload) to assign a specific MAC address to the board. This may have already been done for you since the MAC is generally assigned to the device at the time the rrload bootloader is loaded on the target board. If not, you may use the rrload UI to assign the MAC now and store it persistently in the on-board flash (see Figure 8 on page 36). After connecting the device to a LAN and booting the kernel, you should be able to use the network.

For example, commands like the ones given below should be immediately available to you.

```
# ifconfig
# route
# ping 111.222.333.444
# ...etc...
```

Figure 8.   Network device support (default settings)



Network device support

[✓ y]	-	n	Network device support	Help
y	m	[✓ n]	Dummy net driver support	Help
y	m	[✓ n]	SLIP (serial line) support	Help
y	-	n	CSLIP compressed headers	Help
y	-	n	Keepalive and linefill	Help
y	-	n	Six bit SLIP encapsulation	Help
y	m	[✓ n]	PPP (point-to-point) support	Help
			CCP compressors for PPP are only built as modules.	
y	m	[✓ n]	EQL (serial line load balancing) support	Help
[✓ y]	-	n	DSPLinux LAN ethernet	Help
y	-	[✓ n]	Cirrus Logic Crystal LAN cs8900 ethernet	Help
0x04000000			Base Address for cs8900 ethernet	Help
y	-	n	driver is connected via PIO lines	Help
y	-	n	driver is connected directly	Help
y	-	[✓ n]	SMC 9194 ethernet	Help
y	-	[✓ n]	NE2000/NE1000 ISA ethernet	Help

MOUNTING THE ROOT FILE SYSTEM

By default, the device will **mount** its root file system from an image contained within memory (“root=/dev/rom0”). In this scenario, networking is enabled by the device at boot-time as it utilizes DHCP to learn details about the network to which it is connected. Details such as the netmask, the default gateway, the device IP address, and so forth. This occurs automatically during the kernel boot process.

For convenience, the *kernel command line* assigned to the device through the bootloader’s menu (option 3), in this case, if not explicitly supplied by the bootloader, it can simply be the empty string (" ") since the kernel will resort to the internal default of **root=/dev/rom0**. Recall that the device bootloader lets you pass a custom kernel command line to the kernel as it boots—by default, no kernel command line is needed.

After booting, and then logging into the device, the user may view the network parameters (acquired automatically via DHCP), by issuing the commands below.

```
# ifconfig
```

–and–

```
# route
```

OPTIONAL NFS ROOT MOUNT

As an alternative, the user may decide to set the system up for NFS mounting of the root file system. The idea here is that the user will have a device file system residing on a desktop/server that is mounted (via NFS) directly by the device on boot-up. This is typically done *only* while developing the system. This is because it is easier to experiment with different FS settings without changing installed bits within the device flash. There are two approaches to getting a system such as this operational. (See the two examples below: “NFSroot_#1” and “NFSroot_#2”.)

NFSroot_#1 has the advantage of keeping the kernel command line simple at the expense of requiring your system administrator to add a BOOTP entry in the network table that associates a specific IP address with the specific MAC of your device. On boot, the device sends out a network BOOTP broadcast asking for its IP address assignment.

NFSroot_#1

Note: In this case, BOOTP is used to acquire some network parameters that are already setup by your system administrator.

1. Use **make xconfig** to turn on the following:

```
CONFIG_NFS_FS=y
CONFIG_ROOT_NFS=y
CONFIG_RNFS_BOOTP=y
CONFIG_RNFS_RARP=y
```

2. Rebuild the kernel.

3. Place the device FS somewhere on the network.

4. Boot the device with a kernel command line similar to the command below.

```
"root=/dev/nfs \
nfsroot=192.168.1.15:/local/home/skranz/fs.uclinux \
nfsaddrs=both"
```

NFSroot_#2

Note: In this case, we must use the command line to explicitly assign additional network parameters to the device since, in this case, the device is not using BOOTP to learn those items on boot.

1. Use **make xconfig** to turn on the following:

```
CONFIG_NFS_FS=y
CONFIG_ROOT_NFS=y
```

2. Rebuild the kernel.

3. Place the device FS somewhere on the network.

4. Boot the device with a kernel command line similar to the command below.

```
"root=/dev/nfs \
nfsroot=192.168.1.15:/local/home/skranz/fs.uclinux \
nfsaddrs=192.168.1.125:192.168.1.15"
```

For more information regarding the meaning of the various command line parameters associated with NFS root mounting, please see “linux/Documentation/nfsroot.txt.”

After establishing a configuration that enables NFS root mounting, you need only to...

- Build the linux kernel,
- Build the stock target file system (if you have not already), and then
- Download **BOTH** of them to the board.

You may find it strange that we are downloading the file system to the EVM when our objective is to mount a file system via NFS. Keep in mind that the downloaded file system is used as an intermediate step during the boot process to load the Ethernet module. Once loaded, the system will proceed to un-mount the memory-based romfs and re-mount the root FS via NFS. For convenience this downloaded FS is typically just the stock target FS and most likely the one you are mounting via NFS, however, it does not need to be the same. You could instead download a scaled back FS that simply contains the files below.

```
/linuxrc
/etc/sysconfig/network
/lib/modules/2.0.38.1pre7/rr/ether.o
```

However, forming a special FS like this is likely to be more work than it's worth. Yes, downloading the full-blown stock device FS is much more than the system really needs during an NFS root mount—but it's the easiest way since it makes use of a readily available FS.

Once the boot process loads the ether driver and initializes the networking code, the NFS mount will proceed according to the default kernel command line parameters contained in "`linux/arch/armnommu/kernel/setup.c`" *unless* the NFS parameters are overridden by values passed to the kernel via the rrload bootloader.

10BaseT versus 100BaseT

The default for the C5471 system is 100BaseT, however, using the steps below you can quickly reset it to the 10BaseT operation.

Change to 10BaseT

1. Change the rrload **Makefile** and **rebuild**.

Note: To do this, you must have access to the rrload source code at:

```
dsplinux/c5471/rrload
```

```
NetRate=BaseT100

..becomes..

NetRate=BaseT10
```

2. Change the startup scripts method to **insmod** the ether module and then **download** the new file system.

```
# insmod ether

..becomes..

# insmod ether LAN_Rate=10
```

CHAPTER 6 Rd/Wr Root File System in On-board Flash

The DSPLinux BSP for the C5471 ships with the rrload bootloader, which facilitates the downloading of kernel images and kernel root file system images to the on-board flash.

The default file system used in this BSP is a **romfs**, which is mounted by the DSPLinux kernel as a read-only root file system. While it is true that the DSPLinux boot process also mounts several ramdisk file systems at the root locations **/tmp**, **/var**, and **/root**, it should be noted that data written to these ramdisks at runtime is not persistent; it is lost at the next power cycle. If the root file system files are instead mounted via NFS, then applications can write and/or modify the file system, which is visible at the next power cycle. In this scenario, the root file system is not hosted in the on-board flash but rather on a Rd/Wr hard drive of a remote Ethernet connected machine.

Other areas of the BSP documentation describe more fully the network setup required for mounting a root file system via NFS. As for an on-board hosted root file system, please remember that our reference to its read-only nature refers to the target's lack of ability to write to the file system at runtime. Developers of course have the ability to add (and subtract) from the romfs image prior to installing it on the device. As the device boots up and mounts the image as its root file system, the operation remains read-only for those applications spawned. As mentioned, /tmp, /var, and /root are exceptions.

APPLICATIONS WRITING TO ON-BOARD FLASH

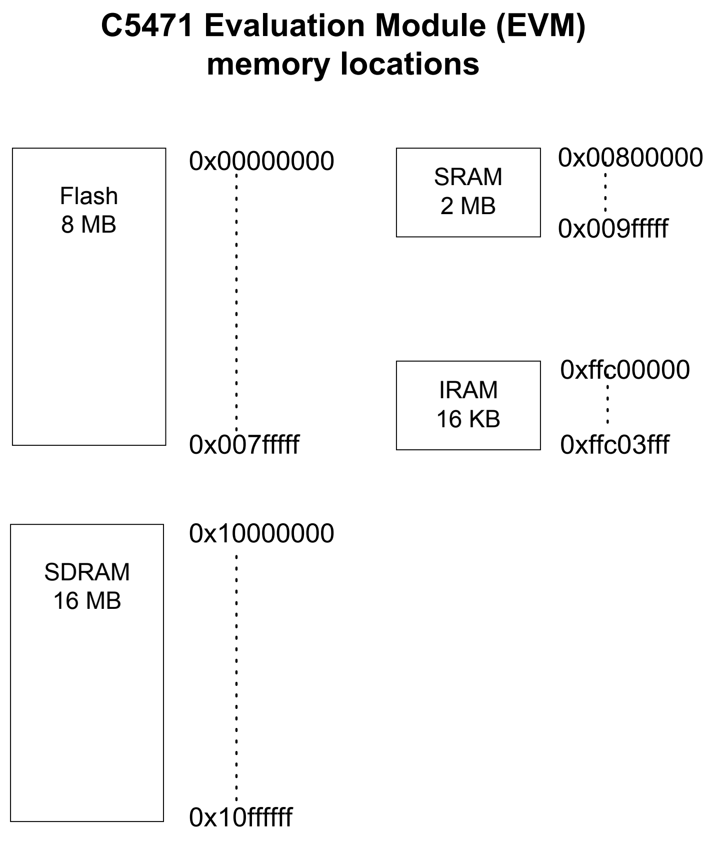
When hosting a file system in on-board flash the lack of persistent writability is acceptable to many embedded systems. However, others will want their DSPLinux applications to store data persistently. For the C5471 platform, these users must extend the current DSPLinux system to allow hosting of an **Rd/Wr** file system within the on-board flash. These extensions are well documented and discussed in the *Linux Open Source embedded systems community*.

As indicated earlier, this requires that you provide a Rd/Wr "root" file system that the kernel can mount on boot-up. It will also require kernel configuration changes (for example, via **make xconfig**) to enable the particular file system type used (JFFS, JFFS2, VFAT..., instead of the read-only romfs currently used), and it requires the addition of the underlying media access layer/driver to support the board's flash chip(s). The DSPLinux BSP provides this for some other platforms, although it is not supplied with the C5471 platform.

CHAPTER 7 DSPLinux Memory Map

The DSPLinux kernel, file system, and bootloader (rrload) make various uses of the available memory areas visible to the GPP processor. These areas fall into those comprised of Flash, SRAM, SDRAM, and IRAM. All of these except IRAM are represented by physical chips external to the main processor. Each area occupies a unique address range, which is discussed in more detail in this chapter. The figure below shows the memory locations of the EVM

Figure 9. C5471 EVM Memory Map



FLASH MAP

The Flash contents are controlled by the board’s bootloader (rrload). Essentially the entire flash space is divided into four specific bins, which the bootloader uses to store the components that it is capable of managing. The bootloader allows each of these to be erased and re-written individually, including the bootloader's own image.

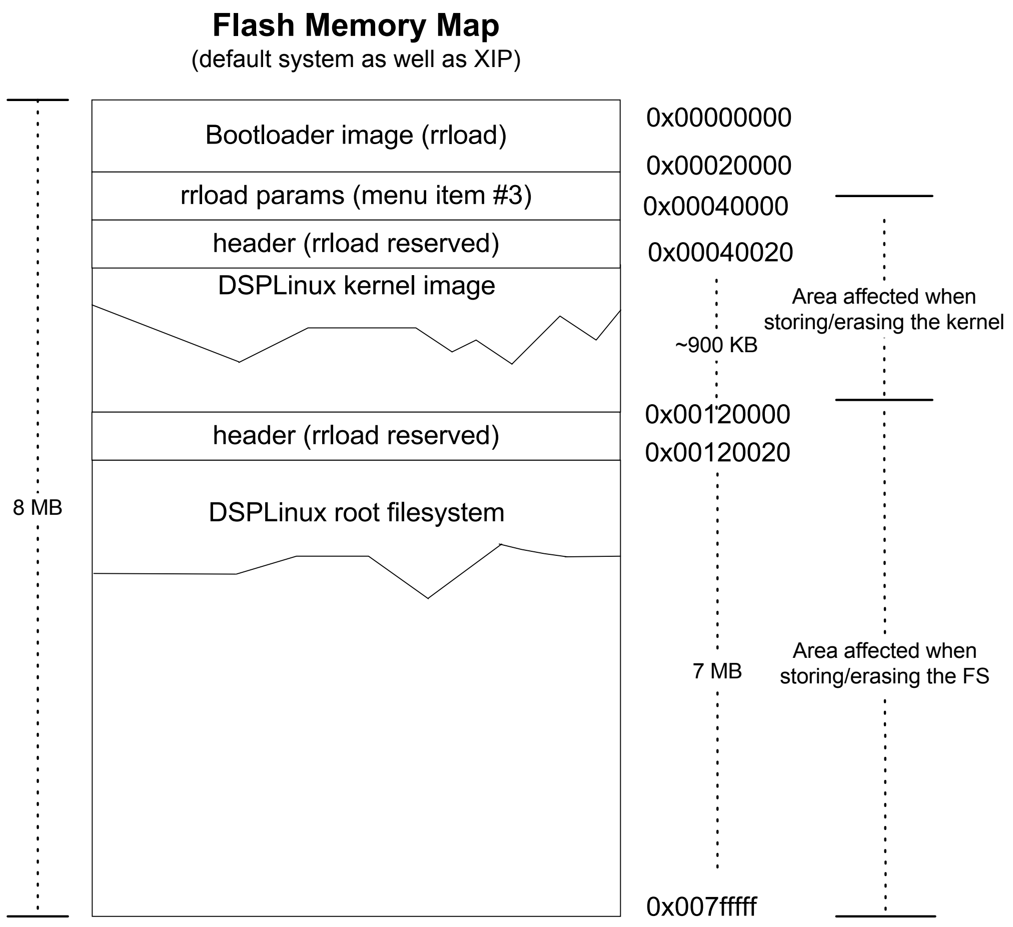
The bootloader *never* runs **XIP** and *always* runs in **SDRAM** allowing it to erase and/or write the flash chips which would not be possible if it was also attempting to execute from within the same chips. At system boot-up, the rrload begins to run directly from flash but then immediately copies itself to SDRAM and transfers control to the copy before proceeding.

The bootloader UI lets you copy image components located in SDRAM into specific areas of flash, including the bootloader itself. This is the method used for getting the bootloader into the flash chips the first time.

- First get the bootloader into SDRAM (via JTAG), and then
- Use it to flash itself as well as any DSPLinux kernel and file system you download to the board (via rrload).

Please consult the rrload section (*see Chapter 11*) for complete details. In addition, if needed, the supplied GDB documentation has a specific example of how to get an instance of rrload installed into flash (see "Installing rrload into a new flash chip" on page 102). Once there, you can bring up the rrload UI by simply holding down the **ENTER** key of the attached terminal window (minicom, hyperterm, and so forth) while simultaneously applying power to the EVM. Otherwise simply apply power to the board and the bootloader will silently execute its user configurable default boot command (usually "boot_auto"), which immediately copies the kernel and file system from flash to SDRAM, and then transfers control to the DSPLinux kernel. See the figure below for a summary of the Flash usage memory map.

Figure 10. Flash Memory Map



IRAM MAP

IRAM begins at address 0xffc00000 and extends to address 0xffc03fff (16 KB total).

DSPLinux uses the high addresses of IRAM as well as the low addresses of IRAM. Specifically, the rrload bootloader uses the **high** *address ranges* of the IRAM space for its *runtime stack*. This stack starts at address 0xffc03f00 and grows to lower address values as the stack is filled. This area is only referenced by rrload until the time it transfers control to the DSPLinux kernel, at which point this area is configured for use as the kernel stack area. Primarily, the kernel makes use of this area during boot-up. Subsequent kernel operations being performed on behalf of the individual application processes will use the individual process stacks contained in SDRAM, not IRAM. Kernel operations being performed outside the context of any particular process will resort to the IRAM stack (for example, networking code responding to incoming packets prior to being delivered to some destination process). The IRAM stack is assumed to be 0x1000 bytes large (range 0xffc02f00 to 0xffc03f00).

The 32 bytes at the **lower** *address range* of the IRAM hold the *interrupt vector table*. This table is loaded as part of the early DSPLinux boot-up, which contains the addresses of the various interrupt handlers assigned to each GPP interrupt. This is necessary since the C547x EVM normally has flash down at the low end of the memory map where the real interrupt vector logic is expected to reside. A side affect is that the kernel can **NOT** plug in its interrupt vectors at run time; the flash being read-only as far as the kernel is concerned.

Instead, we will assume that the bootloader used to invoke the kernel already has a set of very specific interrupt code pre-loaded in flash for the kernel in such a way that it allows the kernel to load a very specific RAM based table (pre-agreed upon between the Bootloader developer and Linux developer). The act of placing the rrload bootloader on the EVM automatically provides this flash logic.

When an interrupt occurs the code at the flash based interrupt location picks up the addresses deposited in the RAM table (by the kernel) then branches to that specific routine itemized in the table.

Summary

The bootloader residing in flash provides code that knows about the kernel's RAM table. Therefore all the kernel must do is insure that the RAM vector table is filled at runtime with pointers to its specific interrupt routines. (See the "linux/arch/armnommu/kernel/entry-armv.S" for details on the DSPLinux kernel side of this agreement.)

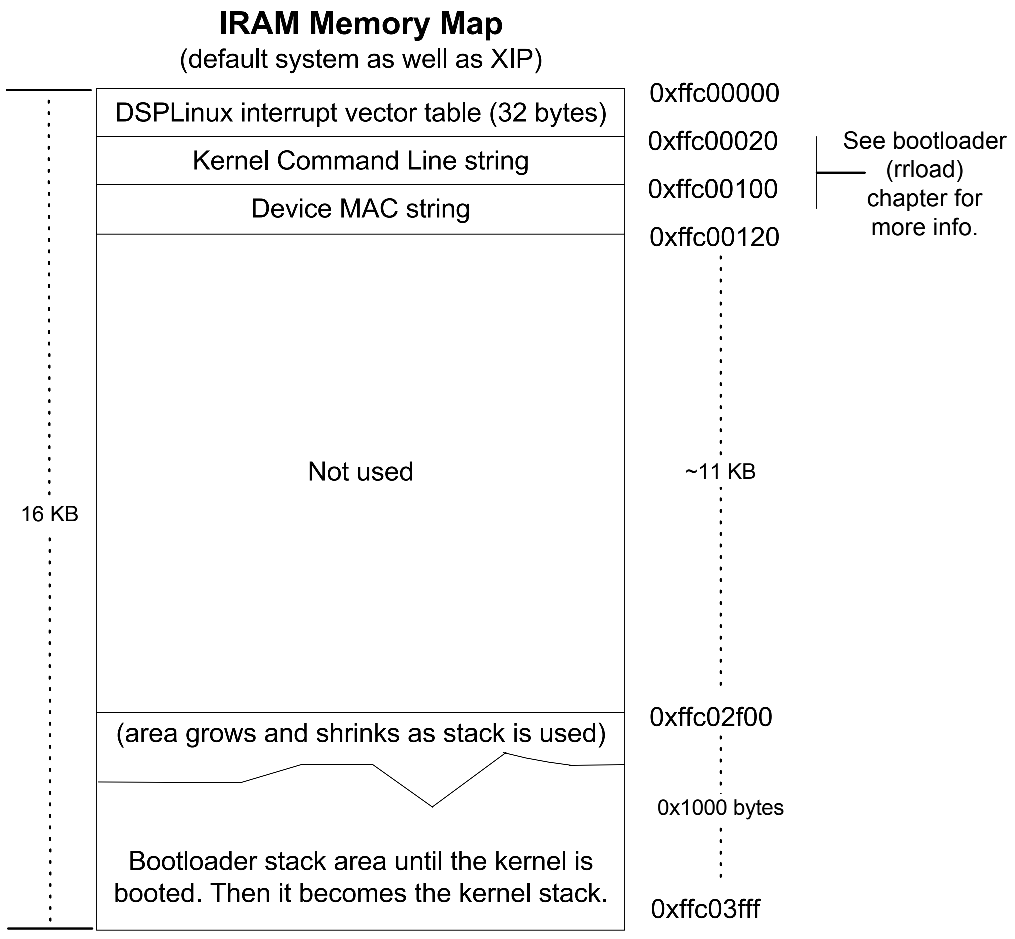
The code snippet below contains the interrupt handler addresses that are copied to the IRAM address 0xffc00000 in the early boot stage of the kernel. However, it is unlikely that any DSPLinux user/developer need be aware of this. It is provided here for clarification and reference only.

```
jump_addresses:

   .word 0 // reset vect not applicable
   .word vector_undefinstr
   .word vector_swi
   .word vector_prefetch
   .word vector_data
   .word vector_addrexcptn
   .word vector_IRQ
   .word _unexp_fiq
```

In addition to these 32 reserved bytes at the low address of IRAM, there are 0x100 additional bytes reserved beyond that. This area is used to hold the Kernel Command Line String and the Device MAC string that the bootloader deposits to this area of RAM and which the kernel subsequently picks up as it is booted. For more information, see Chapter 11 "The DSPLinux Bootloader (rrload)" on page 81.

The IRAM usage memory map is summarized in the figure below.

Figure 11. IRAM Memory Map


IRAM Memory Map
(default system as well as XIP)
DSPLinux interrupt vector table (32 bytes)
Kernel Command Line string
Device MAC string
Not used
(area grows and shrinks as stack is used)
Bootloader stack area until the kernel is booted. Then it becomes the kernel stack.
16 KB
0xffc00000
0xffc00020
0xffc00100
0xffc00120
~11 KB
0xffc02f00
0x1000 bytes
0xffc03fff
See bootloader (rrload) chapter for more info.

SDRAM MAP

The SDRAM address range starts at 0x10000000 and proceeds to address 0x10ffffff (16 MB total). The first several hundred KB (approximately) in this range are **reserved** for the bootloader.

Recall that logic exists at the reset vector (flash memory space) that will copy the bootloader from flash to this area of SDRAM and then transfer control to it. From that point on the bootloader always runs from SDRAM and where it continues to run until it transfers control to some other system such as the DSPLinux kernel—at which point the bootloader is no longer used.

Note: The pad1 and pad2 areas, shown in Figure 12 on page 51, with the bootloader image are there to provide a little extra room in that region. This is so that if a BSP developer wants to move the bootloader stack from IRAM to SDRAM there is room. In addition, if the BSP developer wants to move the interrupt vector table, kernel command line string, or MAC string from IRAM to SDRAM there is some pad space to work with outside the area reserved for the boot loader image.

The bootloader image itself reserves only a 0x20000 range of bytes within the first 0x30000 bytes of SDRAM. By default, the pad areas are unused since the other things mentioned currently reside in IRAM.

The DSPLinux kernel and file system are expected to reside in SDRAM memory following the bootloader's reserved range just mentioned. By default, the DSPLinux build system (makefiles and such) will construct a kernel to load (and run) at an address following the bootloader, while the file system build system constructs a root FS to load at some address following the kernel.

These images arrive in memory the first time by using the bootloader's UI to request the image download from some I/O port (Ethernet, Serial, and so forth). The images are brought into memory at the address indicated by the incoming image (rrbin or srec format). Once you accomplish the download, you can store them to portions of flash used by rrload for holding these images. Once flashed, you can configure the bootloader with a default boot command of "boot_auto" (default). This insures that on each subsequent power up the bootloader automatically copies the images from flash back into the appropriate location of SDRAM prior to transferring control to the DSPLinux kernel.

The location and order of the kernel and the file system stored in flash are independent of the location and order where they are placed in SDRAM. The bootloader manages this for you by keeping the original load/size information with each image so that it can always be copied back to the same SDRAM location that was being used at the time the component was originally downloaded and flashed. You can see the information in the download-ready DSPLinux kernel image produced by the **linux.rr** build process. The first three lines contain the load (size) information that rrload uses to control the place in memory where the image will load.

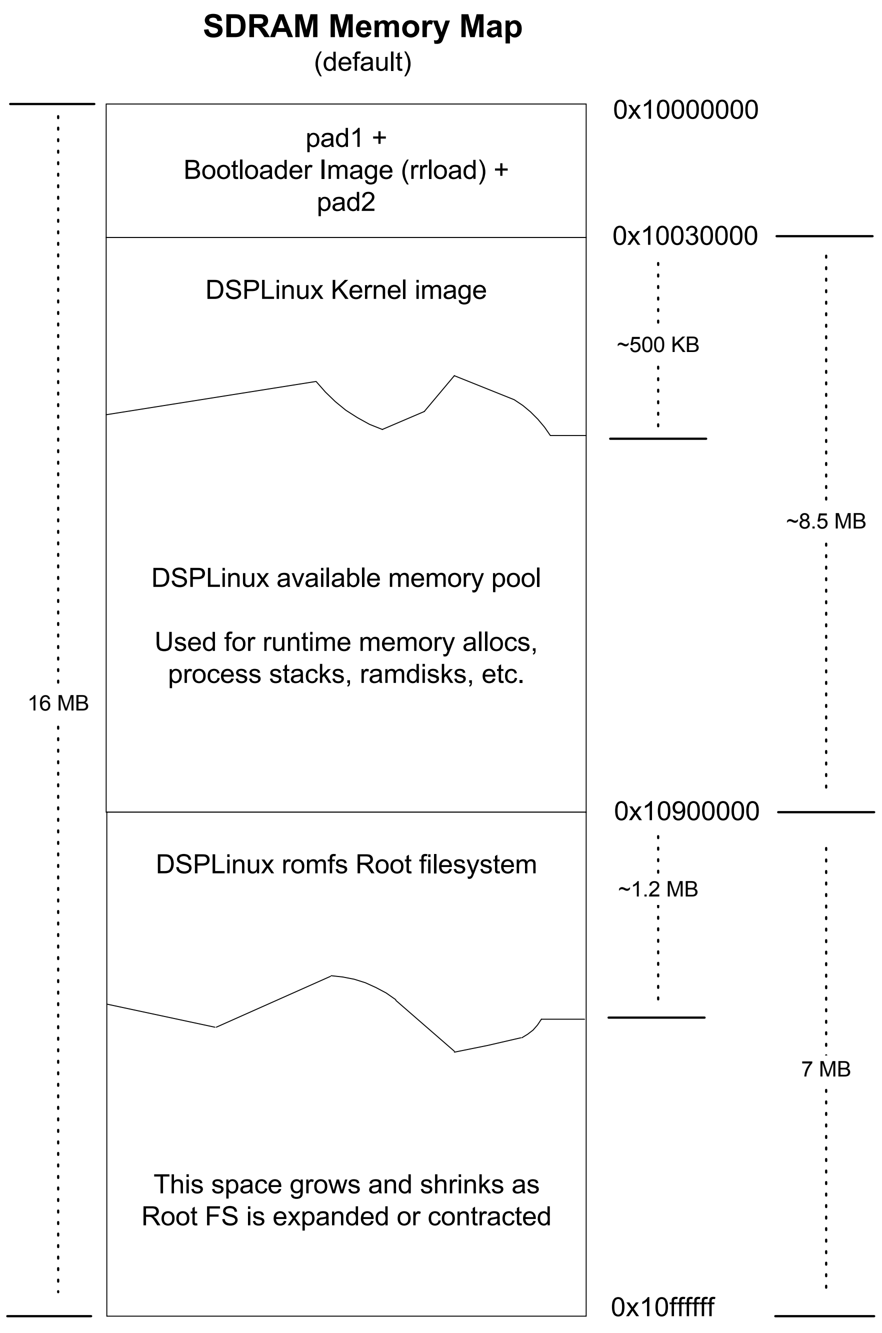
```
$   cd dsplinux/c5471/linux
$   head -3 linux.rr
    >LoadAddr :0x10030000
    >EntryAddr:0x10030000
    >NumBytes :0x00088990
```

Similarly, the root file system image contains information to control where rrload will place it in memory during a download to the EVM. Even when these images are subsequently copied to flash (again, using rrload) this information is maintained to allow rrload to copy them back to SDRAM on the next power cycle.

```
$   cd dsplinux/c5471/linux
$   head -3 romdisk.img.rr
    >LoadAddr :0x10900000
    >EntryAddr:0xFFFFFFFF
    >NumBytes :0x0012F000
```

The SDRAM usage memory map is summarized in the figure below.

Figure 12. SDRAM Memory Map


SDRAM Memory Map
(default)
pad1 +
Bootloader Image (rrload) +
pad2
DSPLinux Kernel image
DSPLinux available memory pool
Used for runtime memory allocs, process stacks, ramdisks, etc.
DSPLinux romfs Root filesystem
This space grows and shrinks as Root FS is expanded or contracted
0x10000000
0x10030000
~500 KB
~8.5 MB
0x10900000
~1.2 MB
7 MB
0x10ffffff
16 MB

SRAM MAP

None of the SRAM memory space is used by the DSPLinux system. You may use this area as you prefer. Recall that you can swap the starting address of this memory area with the FLASH through the runtime register configuration settings (See the chip select setup section in the hardware documents). By default, the SRAM area is found at 0x00800000 and extends to address 0x009fffff (2 MB total).

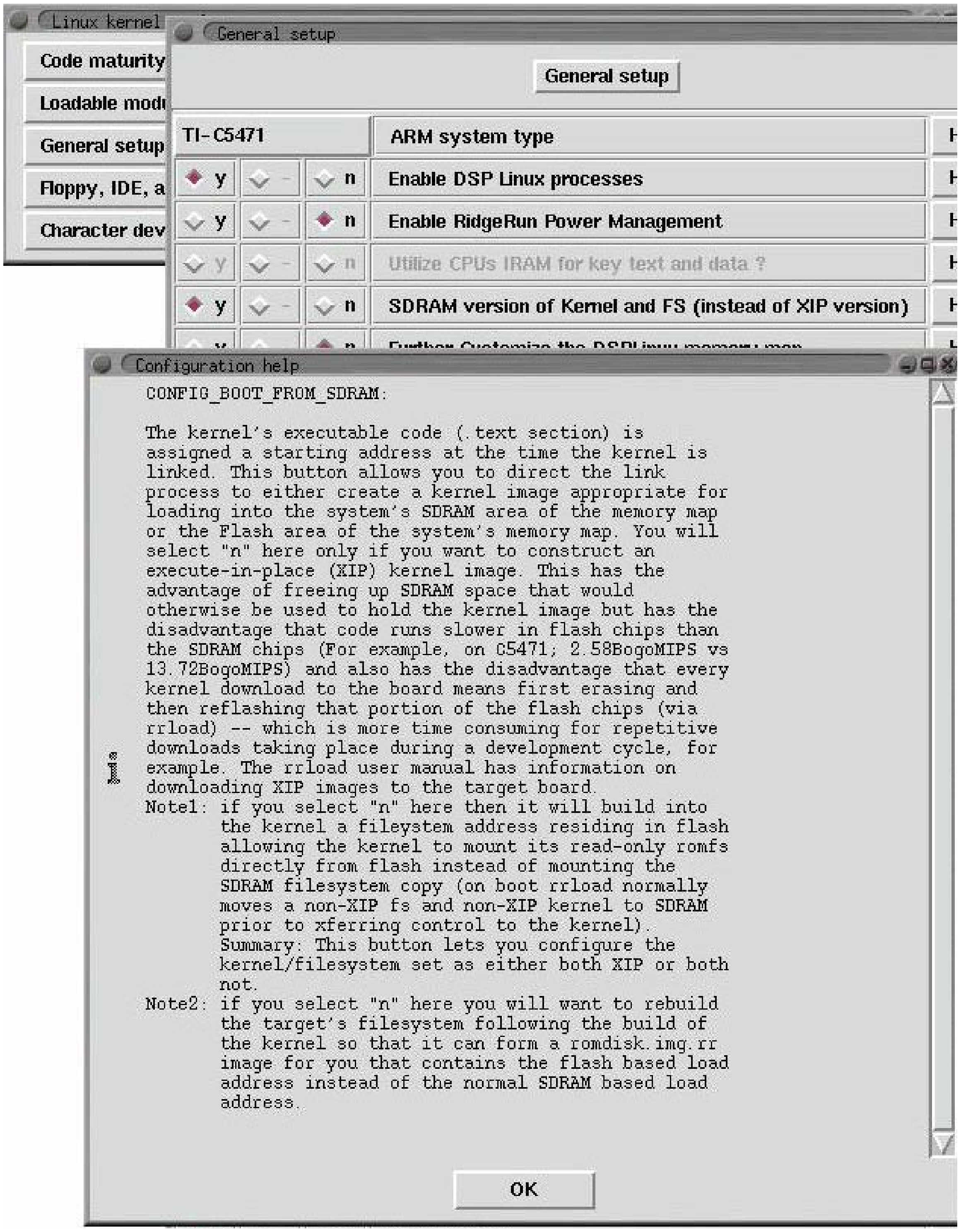
CONFIGURING AN XIP SYSTEM

The default DSPLinux system is intended to run from SDRAM as described above, however, instead you can configure the system so that the file system and kernel are used directly from flash. This is called an **XIP system**, which is configured using the **make xconfig** kernel configuration process.

```
$ cd dsplinux/c5471/linux
$ make xconfig
```

Shown below in Figure 13 on page 53 is the help text associated with the XIP config button from a xconfig session. Please read this and Chapter 11 “XIP images are a special case” on page 93 for information about how to download XIP images to the EVM.

Figure 13. XIP Configuration Help text



TI-C5471			ARM system type
[✓ y]	-	n	Enable DSP Linux processes
y	-	[✓ n]	Enable RidgeRun Power Management
y	-	n	Utilize CPUs IRAM for key text and data ?
[✓ y]	-	n	SDRAM version of Kernel and FS (instead of XIP version)

```
CONFIG_BOOT_FROM_SDRAM:

The kernel's executable code (.text section) is
assigned a starting address at the time the kernel is
linked. This button allows you to direct the link
process to either create a kernel image appropriate for
loading into the system's SDRAM area of the memory map
or the Flash area of the system's memory map. You will
select "n" here only if you want to construct an
execute-in-place (XIP) kernel image. This has the
advantage of freeing up SDRAM space that would
otherwise be used to hold the kernel image but has the
disadvantage that code runs slower in flash chips than
the SDRAM chips (For example, on C5471; 2.58BogoMIPS vs
13.72BogoMIPS) and also has the disadvantage that every
kernel download to the board means first erasing and
then reflashing that portion of the flash chips (via
rrload) -- which is more time consuming for repetitive
downloads taking place during a development cycle, for
example. The rrload user manual has information on
downloading XIP images to the target board.
Note1: if you select "n" here then it will build into
       the kernel a fileystem address residing in flash
       allowing the kernel to mount its read-only romfs
       directly from flash instead of mounting the
       SDRAM filesystem copy (on boot rrload normally
       moves a non-XIP fs and non-XIP kernel to SDRAM
       prior to xferring control to the kernel).
       Summary: This button lets you configure the
       kernel/filesystem set as either both XIP or both
       not.
Note2: if you select "n" here you will want to rebuild
       the target's filesystem following the build of
       the kernel so that it can form a romdisk.img.rr
       image for you that contains the flash based load
       address instead of the normal SDRAM based load
       address.
```

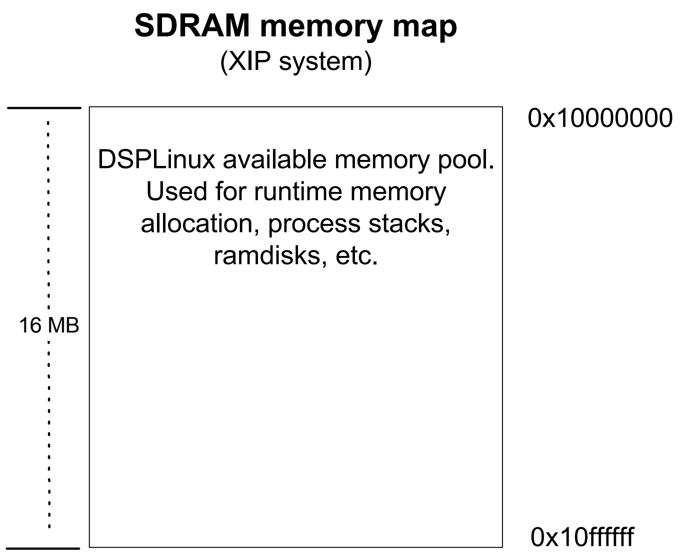
After rebuilding the kernel and the file system notice that the ***.rr** files associated with each component now show a load address within the flash memory range instead of the SDRAM memory range, for example:

```
$ cd dsplinux/c5471
$ . ./setenv
$ # Following builds the file system and the kernel
$ make

$ # The build process created two new *.rr files.
$ cd dsplinux/c5471/linux
$ head -3 linux.rr
    LoadAddr :0x00040020
    EntryAddr:0x00040020
    NumBytes :0x00088990

$ head -3 romdisk.img.rr
    LoadAddr :0x00120020
    EntryAddr:0xFFFFFFFF
    NumBytes :0x0012F000
```

When these images are downloaded to the board (using rrload), the image is then routed directly to the flash memory area that rrload has set aside for holding these components. (The build process is aware of the reserved areas and is careful to construct images having appropriate load addresses). Do not forget to erase the flash areas (using rrload) prior to performing the image downloads. Again, please refer to the rrload documentation in Chapter 11.

The SDRAM usage for an XIP system is shown in the figure below.

Figure 14. SDRAM memory map


SDRAM memory map
(XIP system)
0x10000000
DSPLinux available memory pool. Used for runtime memory allocation, process stacks, ramdisks, etc.
16 MB
0x10ffffff

CUSTOMIZING THE MEMORY LAYOUT

Many users will find no need to customize the system's memory map. For these users either the basic XIP build or the non-XIP build offered by the BSP and discussed in the previous sections will completely meet their needs.

However, the time will come when you are ready to place the system on final product hardware and it is possible that the new hardware will offer a smaller or larger range of SDRAM. Similarly, the platform may offer more or less FLASH than the EVM. In these cases the default settings employed with the BSP's generic XIP and non-XIP builds may not offer the best fit.

Additionally, you may have other reasons for wanting to deviate from the factory settings. For example, if your particular needs require a very large file system and relatively small application memory usage, you may want to re-partition memory to give more to the file system and less to the kernel/apps, and so forth.

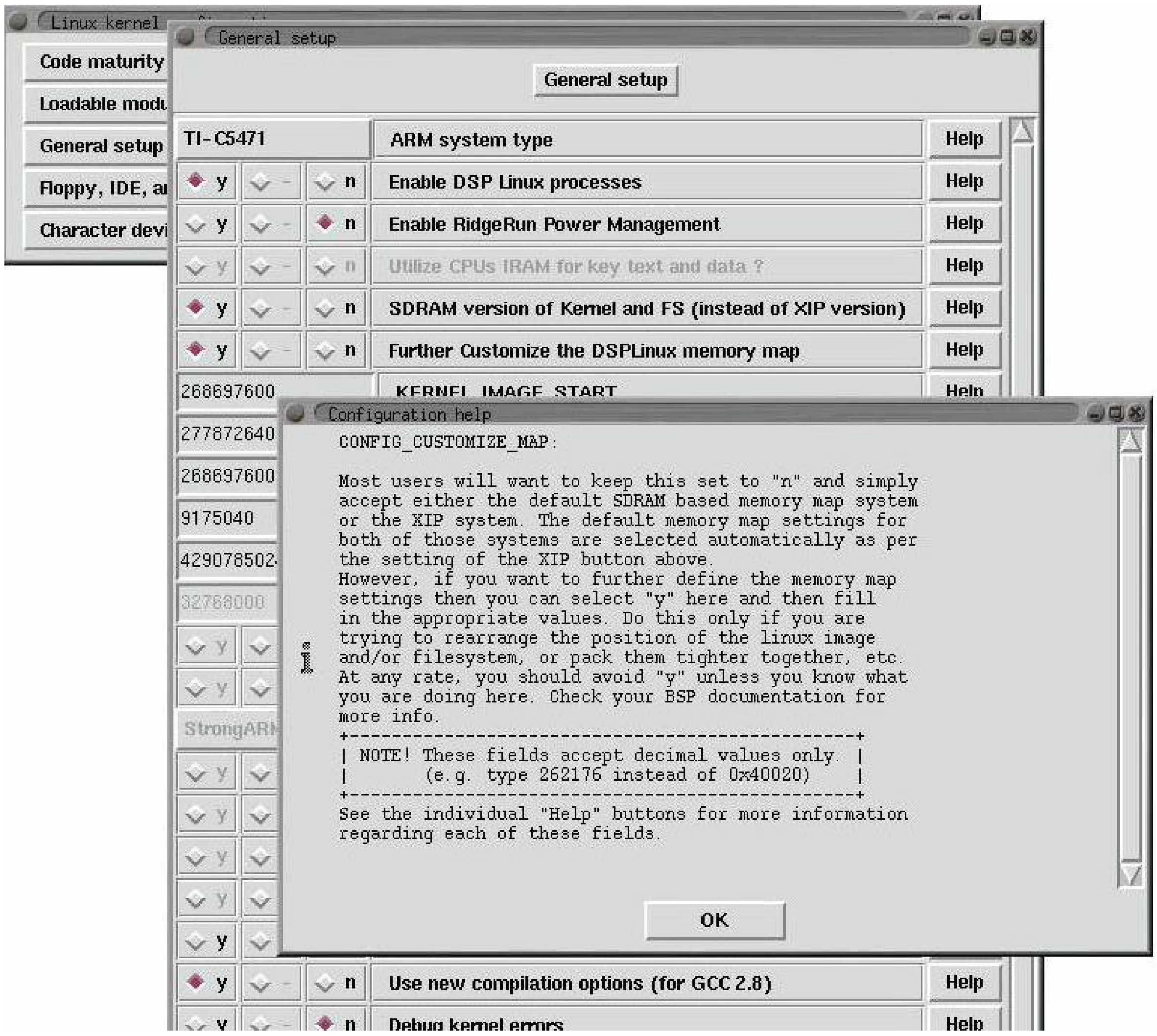
The default factory settings for the XIP and non-XIP system configurations were selected as an estimate of what most users need. For those that want to more finely tune the memory map, the BSP offers several control knobs to use for customization.

Memory Map Control Knobs

```
KERNEL_IMAGE_START
FILE_SYS_START
DRAM_BASE
DRAM_SIZE
KERN_STACK_BASE
```

These knobs are adjustable by using the **make xconfig** process. However, this time instead of simply selecting either an XIP build or a non-XIP build, we will go a step further and request the opportunity to make specific adjustments to the default values normally selected for the above listed items. To do this select "y" for the button labeled "Further Customize the DSPLinux memory map." This is on the General setup panel of the xconfig session and is shown in Figure 15 on page 56 below. In addition, the "Help" text associated with this button is also shown on Figure 15. This enables the custom setting of the five (5) fields that follow it and which correspond to the above listed items.

Figure 15. Xconfig sessions' General Setup



TI-C5471	ARM system type	Help
◆ y - n	Enable DSP Linux processes	Help
y - ◆ n	Enable RidgeRun Power Management	Help
y - n	Utilize CPUs IRAM for key text and data ?	Help
◆ y - n	SDRAM version of Kernel and FS (instead of XIP version)	Help
◆ y - n	Further Customize the DSPLinux memory map	Help
268697600	KERNEL_IMAGE_START	Help
277872640		
268697600		
9175040		
429078502		
32768000		
◆ y - n	Use new compilation options (for GCC 2.8)	Help
y - ◆ n	Debug kernel errors	Help

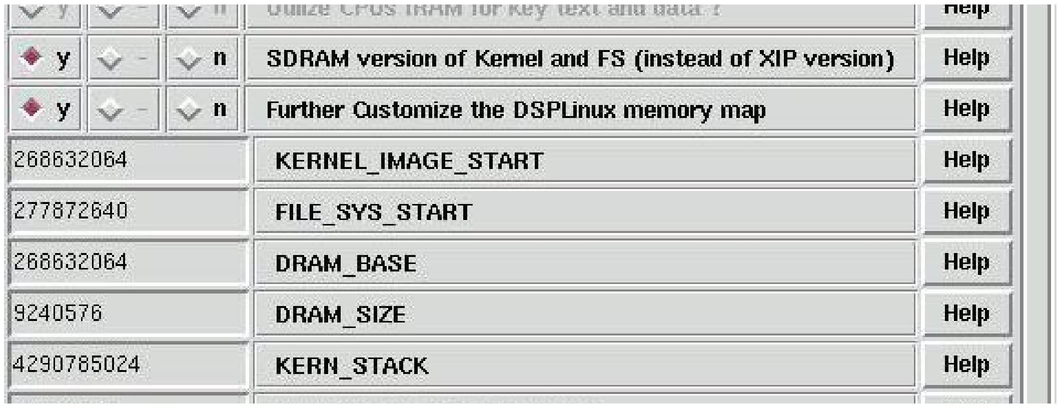
Configuration help

```
CONFIG_CUSTOMIZE_MAP:

Most users will want to keep this set to "n" and simply
accept either the default SDRAM based memory map system
or the XIP system. The default memory map settings for
both of those systems are selected automatically as per
the setting of the XIP button above.
However, if you want to further define the memory map
settings then you can select "y" here and then fill
in the appropriate values. Do this only if you are
trying to rearrange the position of the linux image
and/or filesystem, or pack them tighter together, etc.
At any rate, you should avoid "y" unless you know what
you are doing here. Check your BSP documentation for
more info.
+-----------------------------------------------+
| NOTE! These fields accept decimal values only. |
|       (e.g. type 262176 instead of 0x40020)    |
+-----------------------------------------------+
See the individual "Help" buttons for more information
regarding each of these fields.
```

Having enabled the customization fields, we can now enter our memory map values. These values influence how the kernel is built and how the special 3-line header of the download-ready file system is prepared (romdisk.img.rr). The actual contents of the file system are not affected by these settings, only the information pertaining to where the FS is to reside in memory is effected. Recall that you must rebuild the file system in order to derive this new romdisk.img.rr file.

These settings will control where the kernel will load (run) and they will embed in the kernel the address of the root file system for discovery on kernel boot. This is the reason for recommending that you rebuild both the kernel and the file system after making any changes to the memory map settings. However, it is not necessary to rebuild any applications, libraries, or kernel modules when making these changes. Your settings are made via the fields shown below. Associated with each field is a “Help” button.

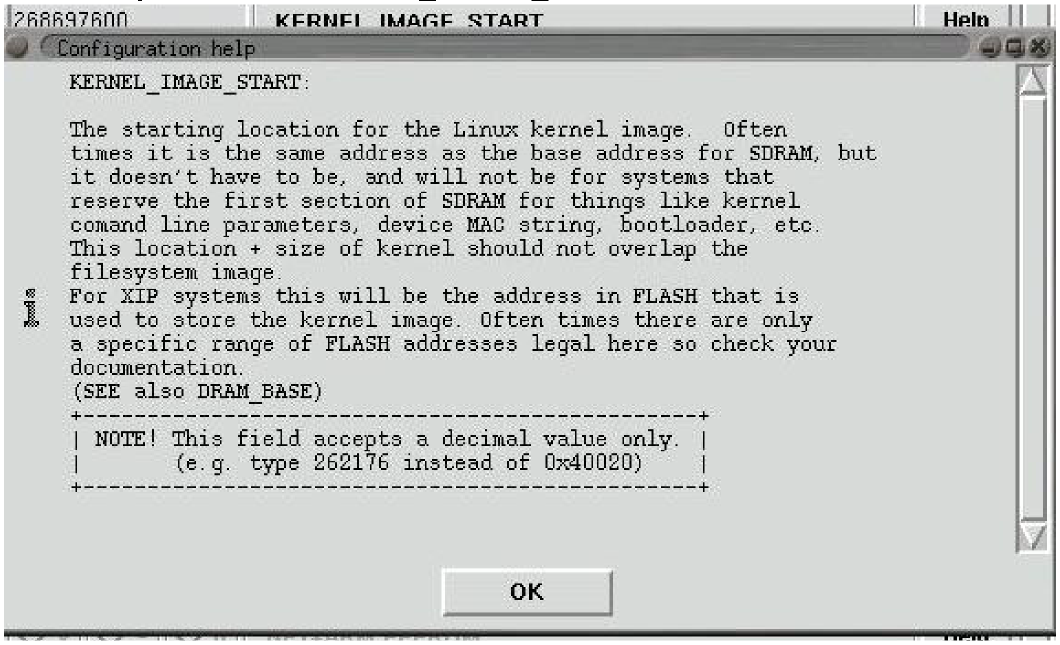
Figure 16.   Help button associated with the General Setup panel



y	-	n	SDRAM version of Kernel and FS (instead of XIP version)	Help
y	-	n	Further Customize the DSPLinux memory map	Help
268632064			KERNEL_IMAGE_START	Help
277872640			FILE_SYS_START	Help
268632064			DRAM_BASE	Help
9240576			DRAM_SIZE	Help
4290785024			KERN_STACK	Help

The “Help” text for each of these fields is presented below along with additional information. We recommend that you double-check the “Help” text for each button while running your xconfig session to insure that you are getting the latest information. If differences exist, the text displayed should be considered more recent than the snapshots shown below.

Figure 17.   Help button — KERNEL_IMAGE_START



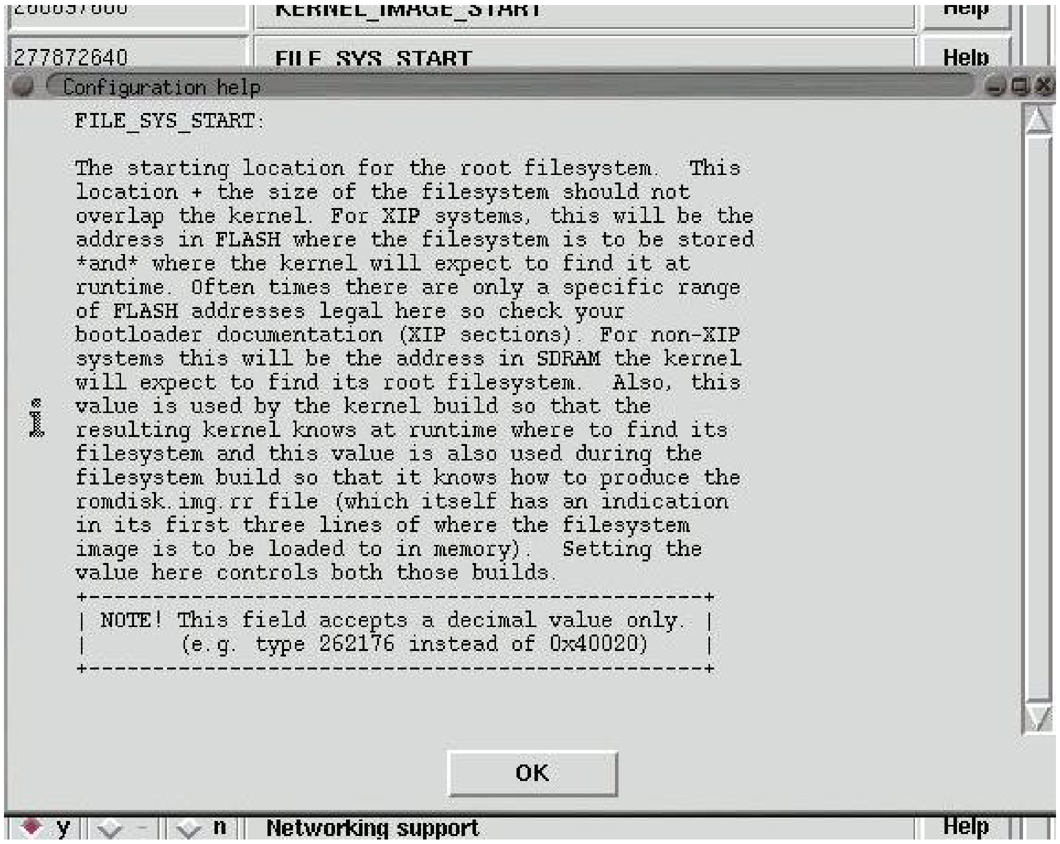
Configuration help

```
KERNEL_IMAGE_START:

The starting location for the Linux kernel image.  Often
times it is the same address as the base address for SDRAM, but
it doesn't have to be, and will not be for systems that
reserve the first section of SDRAM for things like kernel
comand line parameters, device MAC string, bootloader, etc.
This location + size of kernel should not overlap the
filesystem image.
For XIP systems this will be the address in FLASH that is
used to store the kernel image. Often times there are only
a specific range of FLASH addresses legal here so check your
documentation.
(SEE also DRAM_BASE)
+-----------------------------------------------+
| NOTE! This field accepts a decimal value only. |
|       (e.g. type 262176 instead of 0x40020)    |
+-----------------------------------------------+
```

Factory Defaults

```
KERNEL_IMAGE_START268632064 (0x10030000)
KERNEL_IMAGE_START262176 (0x00040020) (XIP)
```

Figure 18. Help button — FILE_SYS_START



```
FILE_SYS_START:

The starting location for the root filesystem.  This
location + the size of the filesystem should not
overlap the kernel. For XIP systems, this will be the
address in FLASH where the filesystem is to be stored
*and* where the kernel will expect to find it at
runtime. Often times there are only a specific range
of FLASH addresses legal here so check your
bootloader documentation (XIP sections). For non-XIP
systems this will be the address in SDRAM the kernel
will expect to find its root filesystem.  Also, this
value is used by the kernel build so that the
resulting kernel knows at runtime where to find its
filesystem and this value is also used during the
filesystem build so that it knows how to produce the
romdisk.img.rr file (which itself has an indication
in its first three lines of where the filesystem
image is to be loaded to in memory).  Setting the
value here controls both those builds.
+-------------------------------------------------+
| NOTE! This field accepts a decimal value only.  |
|       (e.g. type 262176 instead of 0x40020)     |
+-------------------------------------------------+
```

Factory Defaults:

```
FILE_SYS_START277872640 (0x10900000)
FILE_SYS_START1179680 (0x00120020) (XIP)
```

In the XIP case, please note that you are restricted as to which values in the above list you can select. This is because the bootloader, which manages flash, has set aside very specific flash areas each of which will hold one of four components: (1) The rrload image, (2) the rrload user params, (3) the kernel, and (4) the file system.

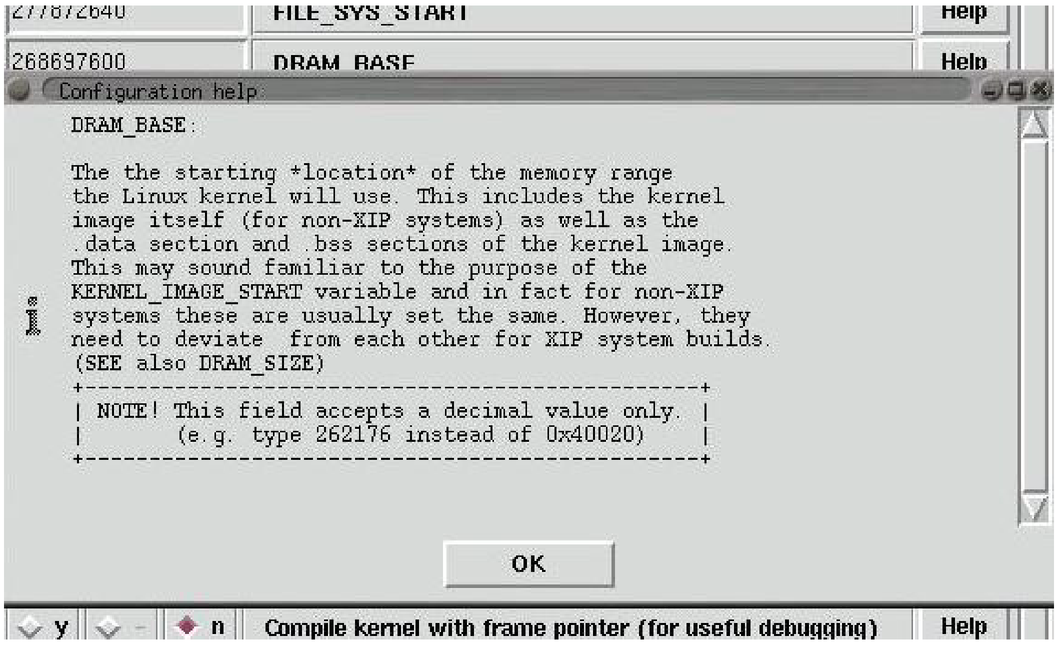
In selecting a custom setting for the FILE_SYS_START (XIP) value, you must select from the legal addresses that the rrload documentation states for this flash area. If the address you assign falls outside the legal range then you need to re-configure the bootloader to define a new flash range and then rebuild it. The re-configure step involves changing the rrload linker script *only*.

The linker script can be viewed at the location below, (for more details, see Chapter 11 “The DSPLinux Bootloader (rrload)” on page 81):

```
dsplinux/c5471/rrload/ld.c5471.script
```

Again, this is not an issue when customizing the FILE_SYS_START setting for a new SDRAM address (for example, non-XIP build). It is only an issue when customizing the FILE_SYS_START with a FLASH address (XIP build) falling outside the rrload’s legal range stated for the file system component.

Figure 19. Help button — DRAM_BASE

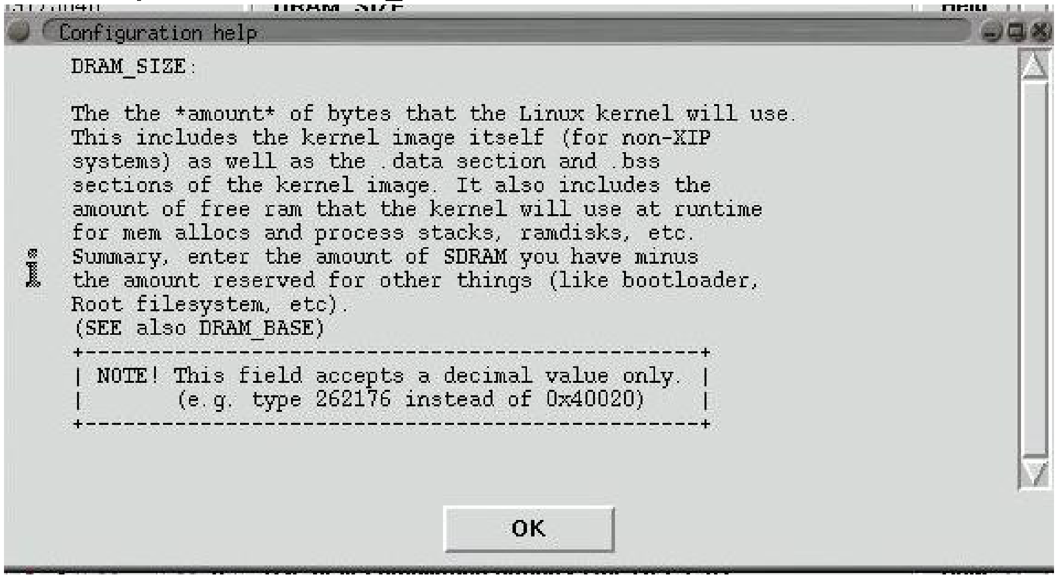


```
DRAM_BASE:

The the starting *location* of the memory range
the Linux kernel will use. This includes the kernel
image itself (for non-XIP systems) as well as the
.data section and .bss sections of the kernel image.
This may sound familiar to the purpose of the
KERNEL_IMAGE_START variable and in fact for non-XIP
systems these are usually set the same. However, they
need to deviate  from each other for XIP system builds.
(SEE also DRAM_SIZE)
+-------------------------------------------------+
| NOTE! This field accepts a decimal value only.  |
|       (e.g. type 262176 instead of 0x40020)     |
+-------------------------------------------------+
```

Factory Defaults:

```
DRAM_BASE268632064 (0x10030000)
DRAM_BASE268435456 (0x10000000) (XIP)
```

Figure 20. Help button — DRAM_SIZE

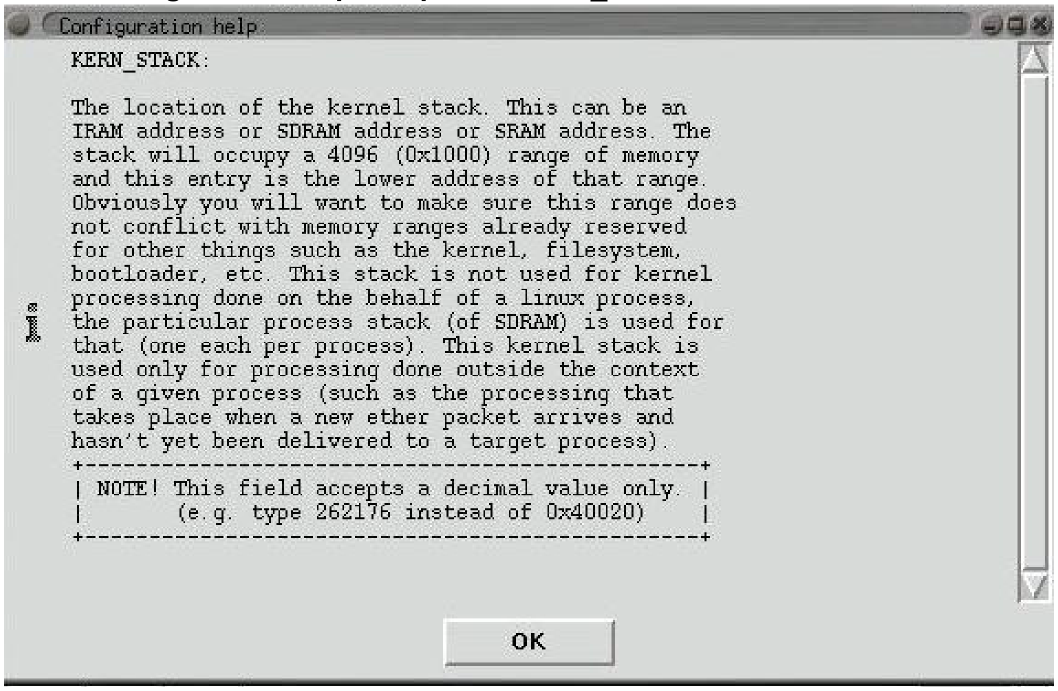


```
DRAM_SIZE:

The the *amount* of bytes that the Linux kernel will use.
This includes the kernel image itself (for non-XIP
systems) as well as the .data section and .bss
sections of the kernel image. It also includes the
amount of free ram that the kernel will use at runtime
for mem allocs and process stacks, ramdisks, etc.
Summary, enter the amount of SDRAM you have minus
the amount reserved for other things (like bootloader,
Root filesystem, etc).
(SEE also DRAM_BASE)
+------------------------------------------------+
| NOTE! This field accepts a decimal value only. |
|       (e.g. type 262176 instead of 0x40020)    |
+------------------------------------------------+
```

Factory Defaults:

```
DRAM_SIZE9240576(0x008d0000)
DRAM_SIZE16777216 (0x01000000) (XIP)
```

Figure 21. Configuration Help sample — KERN_STACK



```
KERN_STACK:

The location of the kernel stack. This can be an
IRAM address or SDRAM address or SRAM address. The
stack will occupy a 4096 (0x1000) range of memory
and this entry is the lower address of that range.
Obviously you will want to make sure this range does
not conflict with memory ranges already reserved
for other things such as the kernel, filesystem,
bootloader, etc. This stack is not used for kernel
processing done on the behalf of a linux process,
the particular process stack (of SDRAM) is used for
that (one each per process). This kernel stack is
used only for processing done outside the context
of a given process (such as the processing that
takes place when a new ether packet arrives and
hasn't yet been delivered to a target process).
+------------------------------------------------+
| NOTE! This field accepts a decimal value only. |
|       (e.g. type 262176 instead of 0x40020)    |
+------------------------------------------------+
```

Factory Defaults:

```
KERN_STACK 4290785024 (0xffc02f00) (non-XIP and XIP)
```

SUMMARY

When it comes to customizing the memory map there are many possibilities. Therefore, if you find that your particular needs cannot be met using the process described above you may want to roll up your sleeves and dive into the internals of the “make xconfig” scripts that control the config session. The particular file that controls the memory map screens presented above is at:

```
dsplinux/c5471/linux/arch/armnommu/config.in
```

If you take a look at the way the current default system settings are taking place, you should get a feel for which few lines of script that you need to change to produce the results that you want. We hope you will not need to do this, however should you find it necessary, start by searching the file for the following lines:

```
(C5471 Control section 1-of-3)
(C5471 Control section 2-of-3)
(C5471 Control section 3-of-3)
```

This takes you to the specific areas that handle control settings for (1) customized configuration, (2) the default non-XIP configuration, and (3) the default XIP configuration.

CHAPTER 8 DSPLinux General Purpose I/O Driver

ARM GPIO

The DSPLinux General Purpose I/O (gio) driver gives the application access to the various GPIO pins of the ARM processor. The EVM has some of these pins already connected to various pieces of circuitry, while others are left unused for you to custom wire should you choose. In any case, this driver will give you the means to control the pin(s) as either an input pin or an output pin. Keep in mind that this is mainly provided for convenience and that the expected usage is only in a **NON**-real-time manner due to issues of attempting to control hardware circuitry from the system's application space.

System response

Normally hardware control is implemented by a dedicated OS driver running in kernel mode. However, for prototype scenarios, you may decide to use the general purpose GIO driver to control LEDS, read the state of a user switch, activate a solenoid, turn on an LCD backlight, or another such **non**-real-time event.

It is not recommended that you attempt to use this driver to create pulse waves of any kind on a given GIO pin. If you have a need to produce (or read) a time constrained pulse pattern on a GIO pin, then it is recommended that you write a specific kernel driver to manage that pattern all within that driver (in kernel mode), since attempting to manage patterns with the general purpose GIO driver will have the system popping in and out of application mode and kernel mode at each pin state transition. In addition, each time the system enters the application mode there is the chance that the Linux system will suspend your application and switch the processor to another system application. By the time the processor is again given to your application, it is possible that an unacceptable time delay was already introduced in the pin-wave pattern that you are attempting to generate.

The C5471 GIO groups and on-board LEDS

The C5471 processor has 20-gio pins corresponding to the GPIO0-19 processor facilities and has another 16-gio pins corresponding to the KBGPIO0-15 processor facilities; two groups of pins, for a total of 36-gio pins. Each of these pins is represented by its own file system device file with names such as **/dev/gio0**, **/dev/gio1**, and so on.

Additionally, the C5471 EVM board has eight (8) LEDS mounted on the board, which can be used by you as you want. You turn these LEDs on-and-off with the GIO driver. Although, the C5471 actually has only 36 (0-35) gio pins, we will present eight (8) more since our gio driver will present the 8-board LEDs as virtual gio pins for convenience.

For example, the User uses the gio driver to open **/dev/gio36** and then writes values to control the board **DS14 LED**; just as if it were connected to a gio pin—which it is not. This means the user does not need to know the board details for gaining access to the eight (8) board LEDs. Instead, we are presenting them as virtual gio pins.

Summary

- `/dev/gio0 to /dev/gio35` are the normal physical GPIO pins of the ARM processor.
- `/dev/gio0 to /dev/gio19` correspond to the c5471 GPIO0 to GPIO19 facilities.
- *`/dev/gio20 to /dev/gio35`* correspond to the c5471 KBGPIO0 to KBGPI15 facilities.
- `/dev/gio36 to /dev/gio43` are the virtual GPIO pins, which the GPIO driver lets you use to control the on-board LEDs

```
/dev/gio36 -- Virtual pin, Use to control LED "DS14"
/dev/gio37 -- Virtual pin, Use to control LED "DS15"
/dev/gio38 -- Virtual pin, Use to control LED "DS16"
/dev/gio39 -- Virtual pin, Use to control LED "DS17"
/dev/gio40 -- Virtual pin, Use to control LED "DS18"
/dev/gio41 -- Virtual pin, Use to control LED "DS19"
/dev/gio42 -- Virtual pin, Use to control LED "DS20"
/dev/gio43 -- Virtual pin, Use to control LED "DS21"
```

Driver system calls

open()

The **open()** system call initializes the driver. Only one program may open each device at a time.

close()

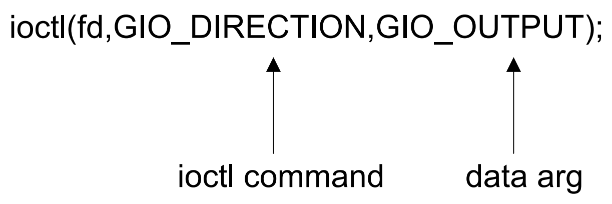
The **close()** system call disables the GIO pin. If an interrupt is connected to the pin then the interrupt handler is uninstalled (see GIO_IRP ioctl below).

Example

```
#include "gio_ioctl.h"
int main (int argc, char **argv)
{
  int fd;
  fd = open("/dev/gio1", O_RDWR);
  printf("fd - %d\n", fd);
  close(fd);
}
```

ioctl()

The **ioctl()** system call is used to configure the GIO pin, read the state of a GIO input pin, and/or set the state of a gio output pin, and so forth. All # defines used in ioctl() calls are defined in the gio_ioctl.h header file. Recall that an ioctl() is comprised of three parameters; (1) a file descriptor, (2) a command code, and (3) a data argument.



THE IOCTL COMMANDS

GIO_DIRECTION

After doing an open() on the pin, configure the pin as either an **output-pin** or an **input-pin**, by using data arg GIO_OUTPUT or GIO_INPUT respectively. Then, see the ioctl() commands below.

Table 1. Input/Output commands

Type	Commands	Description
For output pins...	GIO_BITSET	• If GIO pin is set to output, then this ioctl command will drive the pin **high**. • Data arg not used.
	GIO_BITCLR	• If GIO pin is set to output, then this ioctl command will drive the pin **low**. • Data arg not used.

Table 1. Input/Output commands

Type	Commands	Description
For input pins...	GIO_INVERSION	• Set data arg to GIO_INVERTED or GIO_UNINVERTED. • This applies when reading the state of gio pins and causes the input pin's state to be reported as inverted from the actual value sensed at the input pin. • Does not apply for pins used as outputs. • Default operation is uninverted.
	GIO_BITSET	• If GIO pin is set to input, returns value of GIO pin in data arg. • If GIO pin is inverted the inverted value is reported.
	GIO_BITCLR	• If GIO pin is set to input, same as GIO_BITSET (*see the description above*).
	GIO_IRP	• Set data arg to zero (0) to use pin as an ordinary GIO, no interrupts are used internally for managing the input pin. • Set to one (1) to wire the driver's internal interrupt routine (for the specific gio pin) to either the processor's external IRQ3, IRQ12, IRQ10, or IRQ9—corresponding to gio0, gio1, gio2, and gio3 respectively. • These are the only gio pins with which the GIO_IRP ioctl() can be used. • For these pins, the default is ordinary GIO. • The GIO_IRP ioctl() command is only used when setting up gio pins used for input and if you plan to use the read() call to detect state changes on the pin. • Recall that ioctl(...GIO_BITSET...) is used for reading the pins input state. However, if you are interested in state changes specifically, then use the read() call after setting up the pin with the ioctl(...GIO_IRP...).

read()

The **read()** is used to determine when a state change has occurred on a gio input pin. If the device is opened as O_NONBLOCK, then read returns a value of –EAGAIN if the state change and an associated internal interrupt has not occurred, or zero (0) if it has. If the device was opened normally (blocking), then the read call will block until the input pin state change and an associated internal interrupt occurs. While the state change is the normal way for the call to unblock, a signal sent to the process will also result in a un-block. which is reflected in the return value.

GIO examples

Example 1:

The DSPLinux BSP provides an example application that cycles the board LEDs and shows how the gio driver is used to accomplish this. It can be found at the following locations:

—On the Linux Desktop—

```
dsplinux/c5471/apps/leds/README
dsplinux/c5471/apps/leds/Makefile
dsplinux/c5471/apps/leds/led.c
```

—Files on the Target FS—

```
/examples/leds/README
/examples/leds/led.c
/examples/leds/run_led (runnable script)
```

Example 2:

```
#include fcntl.h
#include asm/arch/gio_ioctl.h
#ifdef DSC21
#define GIO_PIN "/dev/gio26"
#endif
#ifdef C5471
#define GIO_PIN "/dev/gio36"
#endif
int main (int argc, char *argv[])
{
   volatile int i;
   fd = open(GIO_PIN, O_RDWR);
   ioctl(fd,GIO_DIRECTION,GIO_OUTPUT);
   ledstate = GIO_BITSET;
   // Repeatedly flash an LED.
   while (1) {
      if (GIO_BITSET == ledstate) {
            // toggle state.
               ledstate = GIO_BITCLR;
            }
            else {
               // toggle state.
               ledstate = GIO_BITSET;
            }
            ioctl(fd,ledstate,0);
            for (i=0; i<125000; i++) {
               /*silly time elapse*/
            }
         }
         close(fd);
      }
```

Example 3:

```
#include stdio.h
#include stdlib.h
#include unistd.h
#include signal.h
#include fcntl.h
#include sys/ioctl.h
#include linux/types.h
#include asm/arch/gio_ioctl.h
int main (int argc, char **argv)
{
   int fd1;
   int pinstate;
   int status;
   if ((fd1 = open("/dev/gio1", O_RDWR)) < 0) {
      printf("FAIL to open /dev/gio1 - %x\n", fd1);
      exit(1);
   }
   ioctl(fd1, GIO_DIRECTION, GIO_INPUT);
   ioctl(fd1, GIO_INVERSION, GIO_INVERT);
   ioctl(fd1, GIO_IRP, 1); // desire read() calls.
   // get the current state of the pin.
   pinstate = ioctl(fd1, GIO_BITSET, 0);
   switch (pinstate) {
      case 0:
         // pin input value is reported to be low.
         // (actual pin input was high but reporting was
         // inverted as requested).
         break;
      case 1:
         // pin input value is reported to be high.
         // (actual pin input was low but reporting was
         // inverted as requested).
         break;
}
status = read(fd); // potentially a blocking call.
switch (status) {
   case -EAGAIN:
      // no state change yet occurred. This case
      // can only occur if the pin was opened
      // with the O_NONBLOCK flag.
      break;
   case -EINTR:
      // While blocked waiting for a pin state
      // change a signal was received.
      break;
   case 0:
      // Yes, a state change has occurred on
      // the pin.
      break;
   }
   close(fd1);
 }
```

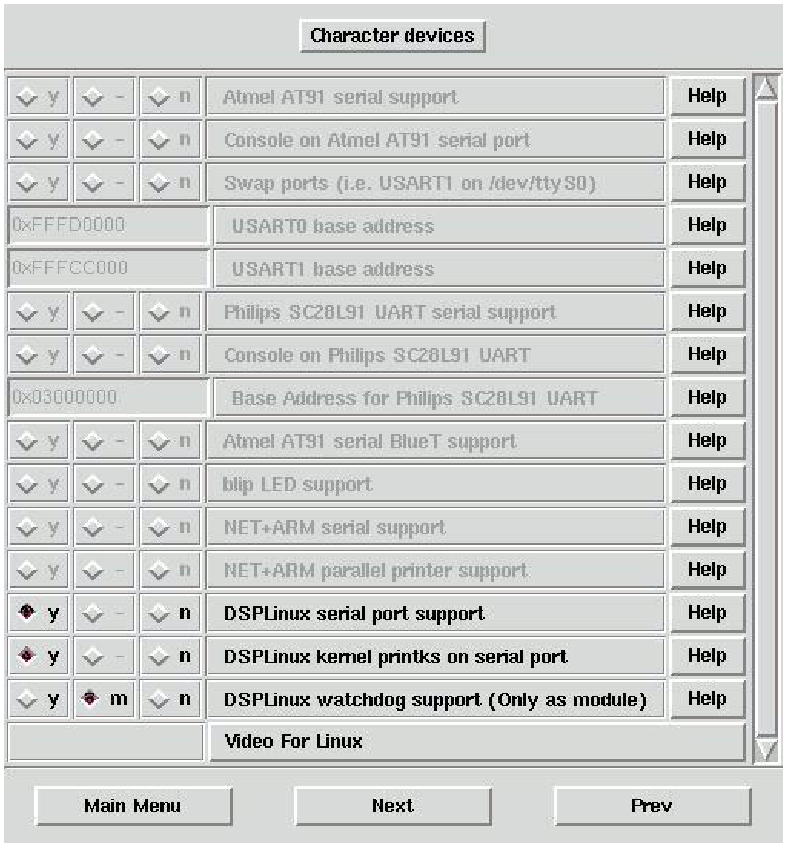
CHAPTER 9 The Serial Port

In this chapter we discuss the serial port and the two linux devices listed below.

- /dev/ttyS0
- /dev/console

The DSPLinux BSP ships with a default kernel configuration that has the appropriate serial driver enabled. Recall that kernel configuration changes are generally performed with the "make xconfig" process as run from the **dsplinux/c5471/linux** directory. With the supplied default configuration, your serial port is immediately available. See to the figure below for a view of the default settings.

Figure 22. Screen shot from a make xconfig session



Character devices

y	-	n	Atmel AT91 serial support	Help
y	-	n	Console on Atmel AT91 serial port	Help
y	-	n	Swap ports (i.e. USART1 on /dev/ttyS0)	Help
0xFFFD0000			USART0 base address	Help
0xFFFCC000			USART1 base address	Help
y	-	n	Philips SC28L91 UART serial support	Help
y	-	n	Console on Philips SC28L91 UART	Help
0x03000000			Base Address for Philips SC28L91 UART	Help
y	-	n	Atmel AT91 serial BlueT support	Help
y	-	n	blip LED support	Help
y	-	n	NET+ARM serial support	Help
y	-	n	NET+ARM parallel printer support	Help
[✓ y]	-	n	DSPLinux serial port support	Help
[✓ y]	-	n	DSPLinux kernel printks on serial port	Help
y	[✓ m]	n	DSPLinux watchdog support (Only as module)	Help
			Video For Linux	

Main Menu Next Prev

PROGRAM ACCESS TO /DEV/TTYS0

With the “DSPLinux serial port support” enabled you can use the **/dev/ttyS0** Linux device to programmatically send and receive characters over the serial port. For reference, the setting has the effect of enabling the CONFIG_SERIAL_DSPLINUX define used during the build of the kernel. An example application is provided that shows how the serial port is used by an application program. On the Linux desktop, the example is found in the BSP’s **dsplinux/c5471/apps/serial/** directory. In addition, within the target files system the runable example is found at **/examples/serial/**.

Example 1: On the Linux Desktop

```
dsplinux/c5471/apps/serial/README
dsplinux/c5471/apps/serial/Makefile
dsplinux/c5471/apps/serial/serial.c
```

Example 2: Files on the Target FS

```
/examples/serial/README
/examples/serial/serial.c
/examples/serial/serial (runable)
```

THE SYSTEM CONSOLE

While **/dev/ttyS0** always represents the serial port, it is not always the case that /dev/console does. Recall that the **/dev/console** represents the system’s **stdin**, **stdout**, and **stderr** file descriptors—but what physical hardware device does /dev/console map too? Is it the serial port, the parallel port, or some display device?

For DSPLinux, when the kernel boots up it first tries to open /dev/console to serve as the console for init. Failing that, it tries to open /dev/ttyS0. Because the default DSPLinux does **NOT** have an internal driver associated with /dev/console the **/dev/ttyS0** becomes the *default console* for all applications.

As a result, if you attach a PC to the target’s serial port and start a terminal session on that PC (such as minicom), then you can login to the remote device via the serial port. This is similar to how a desktop-initiated telnet session lets you log into the remote device via the Ethernet port, except the serial port only supports one user session at a time.

Since the system console is automatically mapped to the **/dev/ttyS0** device, it means that normal **printf()** statements in a program will route out the serial port. Or put generically, any reference to the system’s stdin, stdout, and stderr device descriptors will map to the **/dev/ttyS0**, and therefore the serial port. For example, the command below (entered on the target command line) will send “Hello” out the serial port:

```
# echo “Hello”
```

As mentioned earlier, all programs containing **printf()** statements send output to the system console—hence the serial port. An example program using printf() statements is supplied with the DSPLinux BSP.

Example 3: On the Linux Desktop

```
dsplinux/c5471/apps/hello/README
dsplinux/c5471/apps/hello/Makefile
dsplinux/c5471/apps/hello/hello.c
```

Example 4: Files on the Target FS

```
/examples/hello/README
/examples/hello/hello.c
/examples/hello/hello (runable)
```

Because **/dev/ttyS0** is the default console, applications should **NOT** use it for other purposes. In cases where the board has a second serial port, applications can instead use **/dev/ttyS1**.

BAUD RATE AND OTHER SETTINGS

By default, the DSPLinux device serial port(s) start up at 115200 baud, whereas most host PC’s start up at 9600 baud. The value **115200** *must remain compatible* with the rrload device bootloader, which uses 115200 to maximize serial download speeds. A single terminal session (such as minicom) can communicate with the bootloader and then with the booted kernel without changing any settings. Applications that use a different baud rate can use the standard utilities or /dev/ttyS0 ioctl() calls for changing serial port speed at runtime. The default settings are:

Serial Config

- Standard Cable (not NULL-Modem style)
- 8 data bits
- no parity
- 1 stop bit

KERNEL PRINTK() OVER SERIAL

In the "make xconfig" screen shot shown above in Figure 22 on page 71, you will notice that the "DSPLinux kernel printks on serial port" feature is enabled by default. This is what allows the internal kernel messages to appear over the serial line and therefore on your remote connected terminal session (such as minicom). Recall that **printk()** is **NOT** the same as **printf()**. The setting we use here only applies to the kernel printk() messages. For reference, this setting has the effect of enabling the CONFIG_SERIAL_DSPLINUX_CONSOLE definition used during the build of the kernel.

printk(). . . . . Is only performed within the kernel itself for various diagnostics and OS warning messages.

printf(). . . . . . Is used by applications in user space for general console output.

ADDITIONAL REFERENCES

On Linux desktop

```
apps/serial/*
apps/hello/*
linux/drivers/char/serial_orion.c
linux/drivers/char/tty_io.c
linux/drivers/char/n_tty.c
linux/include/linux/serial.h
```

On target FS

```
/examples/serial
/examples/hello
```

Standard man pages

```
termios(3)
tty(4)
ttys(4)
```

CHAPTER 10 C5471 Power Management

One aspect of power management involves managing the various system clocks. In general, it is desirable to turn the clocks off unless they are really needed and, if turned on, they should be kept at a reasonable frequency since higher frequencies generally mean more power.

The DSPLinux BSP provides a low-level kernel module to assist other driver modules in gaining access to the various clocks that pertain to their specific I/O duties. This low-level power management driver is called **pmck.o**, which is loaded via the **insmod pmck** command. This command is automatically issued at system boot-up from the system startup scripts.

Various DSPLinux drivers are modified to use pmck.o's kernel interface when the power management feature is enabled and built into the kernel. The DSPLinux kernel's default is set with this feature already turned on.

PM manageable clocks

The pmck Power Management module can manage the clocks listed below.

```
arm_clk
spi_clk
i2c_clk
uart_modem_clk
uart_irda_clk
timer1_clk
timer2_clk
watchdog_clk
eim_clk
irq_clk
armio_clk
sdram_clk
audio_clk
arm_pll (supplies clock to mcBSPx, DSP, and TIMER DSP)
```

These clocks are discussed in more detail in the hardware documentation supplied with the EVM board. For more information on these clocks, see the section on the "Clock Management Module" which is found in the "*TMS320VC547x CPU and Peripherals Reference Guide.*"

The **pmck** kernel module lets an I/O driver individually turn these clocks **OFF** or **ON**, and *controls the rate* at which each clock operates.

Additionally, the DSPLinux BSP provides a second kernel module called **pmproc** as an optional companion to the pmck kernel module. It is a loadable module that presents a user interface via a set of **/proc/pm** files, thereby giving access to the underlying "pmck.o" functionality. Command line users or applications that want to tinker with the various clocks will use the "pmproc" kernel module.

PMCK KERNEL MODULE

When the pmck module is loaded (insmod pmck) it automatically turns off some of the clocks. Specifically the clocks listed below.

```
clkm_spi_clk
lkm_i2c_clk
clkm_uart_irda_clk
clkm_timer1_clk
clkm_armio_clk
clkm_audio_clk
```

However, if you do not want pmck to automatically disable the above list of clocks, you can pass it an option when it is loading. Therefore, when the module is loaded the clocks will remain in their current state. For example, to leave clocks in the current state use the pmck option below.

```
# insmod pmck ck_auto_unclock=0
```

PMPROC KERNEL MODULE

The **pmproc** kernel is a loadable module that presents a user interface via a set of /proc files, thereby giving access to the underlying **pmck.o** functionality. The pmproc kernel is not loaded automatically by the system startup scripts. This kernel is provided for those that want to experiment with various clock settings from user space, such as the Linux command line and/or applications.

When pmproc is loaded (insmod pmproc) it creates and populates /proc/pm with files that fall into these three categories.

- Pll rate control (/proc/pm/arm_pll)
- List valid rates in MHZ (/proc/pm/rates)
- Individual clock control (/proc/pm/clkm_*)

Note: Ignore the _25Mhz file in the list below, it is simply a **placeholder**.

The full list of files in the directory are shown below.These files are used in the examples below and on the following pages.

```
# cd /proc/pm
# ls
   _25Mhz              clkm_i2c_clk          clkm_uart_irda_clk
   arm_pll             clkm_irq_clk          clkm_uart_modem_clk
   clkm_arm_clk        clkm_sdram_clk        clkm_watchdog_clk
   clkm_armio_clk      clkm_spi_clk    rates
   clkm_audio_clk
  clkm_timer1_clk
  clkm_eim_clk
  clkm_timer2_clk
```

PLL RATE CONTROL (/PROC/PM/ARM_PLL)

What rate is the PLL generating?

To see what rate a PLL is currently generating, just read its entry. For example, to see the ARM PLL rate from the shell, do the following:

```
# cat arm_pll
50
```

Changing the rate of a PLL

To change the rate of a PLL, simply write the desired rate to its entry in MHz. The driver automatically rounds **DOWN** to the closest possible rate for the PLL. For example, in the case below, the rounding changes the ARM PLL rate to six (6).

```
# echo 10 > arm_pll
# cat arm_pll
6
```

LIST VALID RATES IN MHZ (/PROC/PM/RATES)

To get a list of all supported rates [in MHz] for the PLLs, read the rates entry as follows:

```
# cat rates
6
13
19
25
31
38
44
50
#
```

INDIVIDUAL CLOCK CONTROL (/PROC/PM/CLKM_*)

Below are several examples you can use to disable a particular clock's **WRITE OFF** to its entry or to enable it **WRITE ON**.

Turning Write Off/On

Example 1: Command line

```
# echo off > clkm_spi_clk
# cat clkm_spi_clk
off
# echo on > clkm_spi_clk
# cat clkm_spi_clk
on
```

Example 2: Programatically via an application

```
#include
#include
int main(int argc, char *argv[])
{
   int fd;
   ...
#ifdef CONFIG_RR_PM
   fd = open("/proc/pm/clkm_spi_clk",O_RDWR);
   if (fd < 0) {
      printf("Error: did you forget to `insmod pmproc`?\n");
   }
   write(fd,"on",2);
#endif
   ...
}
```

Example 3: Programatically from a kernel driver

```
   int fd;
   ...
#ifdef CONFIG_RR_PM
   fd = sys_open("/proc/pm/clkm_spi_clk",O_RDWR);
   if (fd < 0) {
      printk("Error: did you forget to `insmod pmproc`?\n");
   }
   sys_write(fd,"on",2);
#endif
   ...
```

Clocks that should NOT *be disabled*

CAUTION: While great care has been put into checking dependencies and rates in the driver, there are settings that can lock up the chip.

Be careful! The list of clocks below are critical. If you disable them, you almost certainly will **lock up**.

```
clkm_arm_clk
clkm_eim_clk
clkm_irq_clk
clkm_sdram_clk
clkm_timer2_clk
clkm_uart_modem_clk
clkm_watchdog_clk
```

CHAPTER 11 The DSPLinux Bootloader (rrload)

What is the DSPLinux Bootloader?

The DSPLinux rrload Embedded Bootloader (rrload), developed by RidgeRun, Incorporated, is part of RidgeRun's DSPLinux BSP. The Bootloader is tailored specifically towards allowing users to manage the loading, storing, and invoking of a Linux kernel and root file system.

In normal operation, the bootloader resides in flash at the reset vector. It is the first program run on power up and, unless intercepted, rrload will typically transfer control to the stored system (for example the kernel + file system). Additionally, the bootloader will relocate either the kernel and/or the file system to SDRAM, if necessary, prior to transferring control. This behavior happens automatically in response to the user having previously configured rrload with "boot_auto" (the default boot command). This is the typical command stored along with the user's other persistent bootloader settings, which among other things can include additional command line arguments that the user wants passed to the kernel.

If the bootloader is not configured with a default boot command, or if the boot process is intercepted, rrload will simply present its user interface and wait for user input. The bootloader offers both a menu user interface (UI) and a command line UI between which the user may easily toggle. Available through the UI is the ability to download a new kernel and/or file system to either RAM or Flash.

The bootloader supports a variety of download formats and can accept these formats over a variety of board I/O ports such as serial, parallel, and ether. The user interface communicates over the board's serial line with a host terminal session such as minicom, hyperterm, and so forth. Holding the **ENTER** key down within the host terminal session while simultaneously applying power to the board, will insure that any previously stored default boot command is temporarily intercepted and forces the bootloader to present its UI. This allows user interaction.

Design objectives

- Intended for embedded systems.
- Easy to build.
- Easy to use.
- Easy to modify.
- Small, simple, and basic.
- Host based UI (minicom, hyperterm, and so forth).
- Offers core bootloader operations; not the kitchen sink.
- Easy to extend and is portable to other platforms.
- Simple and well documented user interface.
- Well documented source code.
- Fresh design and implementation; consistent from front to back.
- Self contained and releasable to the Open Source Community.

Feature set

- Menu UI as well as command line UI.
- UI works with standard host base terminal emulators.
- Can load several image formats to SDRAM including **srec** and **rrbin** (tagged binary) formats.
- Several I/O ports supported; serial, parallel, and ether.
- Flash management utilities built-in for component storage and retrieval.
- User configuration settings survive power cycles (persistent params).
- Automatically executes a list of user-stored commands at boot (default boot command).

Setting up the bootloader

Here we cover what is necessary for the **setup** for the Texas Instruments' C5471/ C5472 platforms. In addition, included in this chapter is information about **persistent configuration** (see "Persistent configuration" on page 85) and the settings that rrload allows the user to customize and store persistently.

Note: If you are interested in interacting with the bootloader's UI and it is not already presented to you on a power cycle, hold down the **Enter** key within your host terminal session while simultaneously resetting the device.

This should intercept the default boot command and force the bootloader to present its UI, allowing user interaction.

Setting up rrload for the C547x EVM

1. With the C5471 or C5472 EVM turned off, connect a serial cable between the EVM and the host PC (use information in the following table).

Table 2. C5471 or C5472 EVM cable type

Cable
Serial cable (male to female, 9-pin, straight through) * ***not*** *a Null Modem type*

2. At the host, bring up a terminal emulator such as minicom and associate it with the particular host serial port to which you connected the cable, using the terminal settings from next table.

Table 3. C5471 or C5472 EVM terminal settings

Settings
115200 baud
8 data bits
1 stop bit
No start bit
No parity
No flow control

3. Change the jumpers listed in the table below to enable them to **boot-from-flash** based on the positions in the “Move to...” column. All other jumpers use the stock-board jumper settings.

Table 4. C5471 or C5472 EVM jumper positions

Jumper	Move to...
JP21Big/Little Endian	2-3 position
JP27RAM/FLASH Swap	1-2 position
JP28FLASH WE Disable	1-2 position
JP29ROM size Select	left-middle position

4. Assuming that the board has rrload contained within on-board flash and the board switches/jumpers are enabled for boot-from-flash, then **apply power** to the *target board*.

   As the board powers up you should see the rrload UI appear in the host mini-com session window. The bootloader now has control of your target and is waiting for user input.

PERSISTENT CONFIGURATION

Option 3 in the main menu shows a list of current user settings. You can view these same settings from the command line mode by issuing the “set” command. If you make changes to these settings, it only affects the current session (RAM-based settings). If you want the *new settings to survive power cycles*, you need to re-flash them by first performing an erase of the params section of flash (the RAM copy is unaltered), followed by a **flash store** of the new params causing the current RAM settings to be copied to flash. You can perform this from menu mode or command line mode.

Below is an example showing the setting (storing) of a new bootloader default boot command. In this case, as an alternative to using the menu mode UI, we are using the bootloader’s command line UI to assign the **bootcmd** parameter. Here we assign an empty string since no value string was supplied on the first line.

```
rrload> set bootcmd
rrload> eraseflash params
rrload> copy -c params -s ram -d flash -f na
```

Below are the settings that rrload allows the user to customize and store persistently.

- **Default UI mode**
  The “mode” Valid settings are: cmd, menu. This allows the user to establish which UI the bootloader will present by default.
- **Default boot command**
  The “bootcmd” is any list of bootloader commands. Use ‘;’ to separate your commands. For example you might want to use this to have the bootloader automatically perform a (1) kernel xfer to ram, a (2) file system xfer from flash to ram, and then (3) boot the kernel.

  This example could be accomplished with the following default boot command: copy -c k -s f -d r -f na; copy -c f -s f -d r -f na; boot (Although it should be noted that this particular example could be accomplished even more easily with a single “boot_auto” command.)
- **I/O load format**
  The “loadfmt” Valid settings are: srec, rrbin
- **I/O load port**
  The “loadport” Valid settings are: ser, par, ether. When the load port is set to ether then the other parameters pertaining to TFTP are enabled. Otherwise they are ignored.

- **server IP**
  The “serverip” is the IP of the remote server that delivers the TFTP based image.
- **server MAC**
  The “servermac” is the MAC of the remote server (TFTP).
- **device IP**
  The “devip” is the IP of the device which will initiate and receiving the TFTP based image during an ether transfer.
- **device MAC**
  The “devmac” is the MAC of the device (TFTP).
- **Kernel Path**
  The “kpath” is the path name of the kernel that is to be xferred from the remote server to the device over the ether port using TFTP.
- **FileSys Path**
  The “fpath” is the path name of the file system that is xferred from the remote server to the device over the ether port using TFTP.
- **Kernel Parameters**
  The “kcmdline” is the command line string that is passed to the kernel as it is being invoked by the bootloader.

Note: As mentioned earlier, the user can change any of these SDRAM based values, however, the new values will remain in effect **ONLY** during the current session. If the user wants the new settings to survive power cycles then the user must take steps to erase the params section of flash and re-store the params to flash. On boot-up, the bootloader will always insure that the SDRAM based params are initialized to the flashed set.

DOWNLOADING A NEW KERNEL AND/OR FILE SYSTEM

One of the main reasons the bootloader exists is to give the user a means to load images onto the EVM board. Specifically, we are interested in loading a kernel and possibly a root file system. Without the bootloader the user is restricted to JTAG based downloads which can be an attractive option. However, not all users have the supporting tools necessary to facilitate JTAG based development. The rrload bootloader gives the user an immediate path to loading and running software components on the board. Additionally, the bootloader allows the user to store the components to flash if desired.

The bootloader supports a variety of download image formats, which can be streamed through the various I/O ports of the EVM. The architecture supports the ease of adding more modules to support additional image formats or I/O ports as necessary. Currently the two image formats and three I/O ports below are supported.

Image Formats

- Motorola® **srec** format
- RidgeRun **rrbin** format.

I/O Load Ports

- Ethernet Port (via TFTP)
- Serial Port
- Parallel Port (incomplete)

Creating an srec image

The **srec** format is a Motorola inspired binary standard format. This ASCII format is used for downloading executable files, not data files. An srec file is typically created prior to download using the following command where myprog is a standard executable file format such as ELF or a.out:

```
$ arm-uclinux-objcopy -S -O myprog myprog.srec
```

Creating an rrbin image

A RidgeRun inspired format, **rrbin** is a tagged image binary format used for downloading either executable files or data files such as file systems, and so forth. This format is very efficient and can be generated with the help of the RidgeRun supplied utility called **mkimage**, which is used to prepare a file for download to the rrload bootloader.

Note: The **--LAddr** used below is just an example, the actual value you use is dependant on the address where you need load your specific object. For example, in the case of a kernel, you could use a command such as **arm-linux-objdump -h linux** to find out where the image expects to load. In the case of a file system the **--LAddr** value should match the address where your kernel expects to find the file system during a kernel boot. Use **mkimage** to prepare both kernel and file system images for download.

```
$ mkimage --LAddr 08e00000 linux linux.rr
```

– or –

```
$ mkimage --LAddr 08c00000 romdisk.img romdisk.img.rr
```

Note: In the example below, the `LoadAddr` was determined by the user options supplied (`--LAddr`), while the `EntryAddr` and `NumBytes` are computed by the `mkimage` script. In cases where the input image does not represent an executable, but rather represents a file system [for example], then the `EntryAddr` will not apply and `mkimage` will simply supply a default filler value. For those that are curious, there is additional usage information contained within the document header of the `mkimage` script.

The resulting ***.rr** file is a space efficient binary format with an additional header tacked on the front to allow the rrload bootloader to accept the file with a minimum of coupling between the rrload source code and the software components downloaded to it. This special header is what makes it a tagged image format. It contains the following pieces of information used by rrload to process the enclosed pure binary data. Below is an example of the internal data of a ***.rr** file.

```
>LoadAddr:0x08e00000
>EntryAddr:0x08e00130
>NumBytes:0x0000a200

(binary block: 0xa200 bytes)
```

When faced with the choice of creating an srec image or rrbin image, rrload users generally opt for the **rrbin** format for both the kernel and file system components. Once the images are created, they can be downloaded to the EVM using UI menus in conjunction with the prior settings made to rrload parameters (main menu, option 3). In particular, you will want to adjust the bootloader configuration to indicate the I/O port and image format you will use. However, when using the bootloader's command line mode, the "copy" command contains all the information needed to direct the download.

Command-line mode examples and descriptions

Note: The command below only loads the kernel, it does **NOT** cause the kernel to start running. The bootloader's **boot** and **boot_auto** commands are the only way to transfer control to a kernel.

```
rrload> copy -c kernel -s serial -d ram -f rrbin
```

This command copies the kernel from the serial port to a RAM destination using the **rrbin** input parser. The incoming file determines to which RAM addresses they are written; remember the rrbin header shown above? The **LoadAddr** field of the incoming file determined to where in the system's memory map the image is loaded. The image was built specifically to load and run at this address.

The header information is also acquired by rrload when processing incoming **srec** images. The bootloader keeps this information with the image, therefore it always knows where in memory it needs to reside when used. This will come into play if the user subsequently requests that the bootloader stores the kernel to flash. How does the bootloader know what a kernel is? It does not really, except in this command the "-c" kernel informed the bootloader that this image is a "kernel." Hence, when the user later requests a kernel boot, or kernel store, and so on, the bootloader has a record to what image the term "kernel" corresponds.

```
rrload> copy -c k -s s -d r -f r
```

This does the same thing as the command in the prior example above.

Note: The command below requests the bootloader to pull in the **rrbin image** using the ether port. This works only if the bootloader was previously setup with the correct TFTP related user configuration. For more details, see "Ethernet downloads" on page 95.

```
rrload> copy -c filesys -s ether -d ram -f r
```

The incoming **rrbin** file system image is copied from the ether port to the memory map RAM locations which the image calls. Like kernel images, the bootloader maintains a record with this image indicating the location in memory to where it originally loaded. If the user requests that the file system be moved to flash, and later power cycles and requests the file system be moved to SDRAM, the bootloader will have the necessary information to place it in the correct location.

```
rrload> set
```

Shows the current bootloader user configuration settings.

```
rrload> help
```

Provides additional information regarding the "copy" command and others. Also shows how you can change the bootloader user-config settings if necessary.

```
rrload> copy -c k -s e -d r -f r; copy -c f -s e -d r -f r; boot
```

A command list requesting a kernel and file system download, followed by a boot of the kernel. The "boot" command transfers control to the last kernel downloaded, or if there wasn't one, it will attempt to locate one stored in flash and transfer it to the **EntryAddr** recorded with that kernel image.

```
rrload> copy -c f -s e -d r -f r; copy -c k -s e -d r -f r; boot
```

This command is exactly the same as the command above except the order of the loads are reversed. (First the file system, next the kernel, and last boot the kernel.)

Note: In the next three commands, we have supplied the "-f na" because we are moving a pure memory image from flash to SDRAM and it does not make sense to state "-f srec" or "-f rrbin" since the images are neither of those formats; this is just a straight memory copy.

```
rrload> copy -c kernel-s flash -d ram -f na

rrload> copy -c filesys-s flash -d ram -f na

rrload> boot
```

Moves the kernel and file system stored in flash to SDRAM according to the **LoadAddr** recorded with it. Then boots the kernel by transferring control to the **EntryAddr** recorded with the kernel image.

```
rrload> boot_auto
```

This single command is exactly the same as the 3-command-set shown above. This is typically the command the user assigns to the bootloader's default boot command so that on power-cycle the kernel automatically boots. This is the command mapped to option 4 in the main menu.

```
rrload> set bootcmd boot_auto
```

This command sets the bootloader's default boot command to a string that represents the command (or command list) the user wants the bootloader to automatically execute on each power cycle. Using this type of command, a user can set things up so that the kernel boots directly on a power cycle instead of presenting the bootloader's UI. However, holding the **ENTER** key down within the host terminal session while simultaneously applying power to the board will insure that any previously stored default boot command is temporarily intercepted and instead forces the bootloader to present its user interface.

Note: For documentation convenience, the examples above where performed using the bootloader's command line UI. However, you can perform all of these operations equally well using the bootloader's menu UI.

STORING A NEW KERNEL AND/OR FILE SYSTEM IN FLASH

As mentioned earlier, one of the main purposes of the bootloader is to facilitate getting software components loaded onto the board (via I/O ports). Another main purpose for the bootloader is to facilitate the storing and retrieving of software components within on-board flash. Specifically, the rrload bootloader was designed to manage four such component types:

- Bootloader code image
- Bootloader user config settings
- Kernel image
- File system image (for kernel)

Each of these components has a very specific area of Flash dedicated to holding it. There can *only* be one (1) such instance of each component within flash at any one time and, prior to replacing any content, the user must explicitly erase the previous component content. The rrload UI provides the ability to individually erase and store the four components listed above. The bootloader image area of flash is typically not something a person would deal with unless performing the steps associated with installing a copy of rrload on the EVM board. The bootloader's user configuration area of flash holds the various configuration settings available from the main menu's Option 3. However, the kernel and file system areas of flash are what we want to talk more about in this chapter.

Kernels and file systems can be stored two ways...

1. By using the store functions available from option 2 of the main menu to copy content existing in SDRAM over to flash. Alternatively, you can use the "copy" command issued from the bootloader's command line UI to store content into flash. In upcoming discussions, we refer to this as the **RAM-based** case.

   — or —

2. By performing a kernel or file system download via an I/O port where the incoming image indicates load addresses that are in the flash memory map area of the system. For example, a kernel built to execute in place (XIP) within the flash is linked to load from an address within the flash portion of the memory map and therefore an rrload download of that image routes the file directly to flash. We consider this a **flash-based** image.

The RAM-based case

In the **RAM-based** case the user can remain unaware of the flash addresses used to store a kernel and file system. The bootloader takes care of tagging the stored component with enough information to allow the user to later pull this component back out of flash and into its original SDRAM location prior to use. For example, the steps below will load a **RAM-based** image and store it to flash.

```
rrload> copy -c kernel -s ether -d ram -f rrbin
rrload> eraseflash kernel
rrload> copy -c kernel -s ram -d flash -f na
```

If the current SDRAM contents are lost (for instance by a power cycle), the user can retrieve the kernel from flash and boot it with the commands below.

```
rrload> copy -c kernel -s flash -d ram -f na; boot
```

Note: At no time must the user be aware of the specific addresses used for storing the component in either flash or SDRAM. This is true of all 4-component types. Of course, there always is an exception and, in this case, it is revealed during the **flash-based** image case discussed next.

XIP IMAGES ARE A SPECIAL CASE

The flash-based case

As we mentioned earlier, the **flash-based** case is unique in that an image is written directly to flash due to the downloading of a kernel or file system via an I/O port. This occurs when the image contains load addresses that fall within the flash area of the system's memory map. The address is assigned at the time that a **linker** produces an image and then **mkimage** prepares it into an rrbin image.

This falls in the realm of advanced usage and requires some special attention because mistakes in this area can corrupt the board's flash content to the point of not permitting a boot. In this case, the user is forced to install a new copy of rrload using the JTAG connector.

The critical area pertains to the selection of the load address assigned to the image. A user is **NOT** free to locate the image wherever they choose, because areas of the flash are already set aside for the bootloader's use and are *reserved*. Instead, the user is limited to a sub-range within the flash for placing XIP kernel images and another specific sub-range for placing XIP file system images. Loads performed to these specific ranges are managed by rrload by way of tagging the loaded image with the normal record keeping information normally associated with kernels and file systems stored to flash—just as when normally storing components (copy from SDRAM to flash).

This record keeping information is stored automatically by rrload during the download and contains enough information so that rrload knows this is an XIP component. For example, the following command would not result in a copy to SDRAM since rrload can see that the stored kernel did not originate from SDRAM and instead was loaded by XIP. The requested copy is ignored and the kernel is simply booted in place.

```
rrload> copy -c k -s f -d r -f na; boot
```

The steps for performing an XIP download are rather straightforward, however, it must be emphasized that it assumes that the user took care to build the load image with a valid load-address.

If you send an image to the wrong area of flash, you will most certainly corrupt data reserved for the bootloader's use. As always, we erase the component's flash area first before placing new content there. These are the steps to follow:

```
rrload> eraseflash -c kernel
rrload> copy -c k -s e -d r -f rrbin
rrload> eraseflash -c filesys
rrload> copy -c f -s e -d r -f rrbin
```

Note: You might expect the **-d** field to be **-d flash** instead of **-d ram**; it turns out that either one will work.

Important! The steps above are only successful if you build the XIP kernel and XIP file system to be fully contained within the following special flash address ranges:

Flash address ranges

```
XIP kernel:Flash range 0x00040020 to 0x0011ffff (inclusive)
XIP Filesys:Flash range 0x00120020 to 0x003fffff (inclusive)
```

SERIAL DOWNLOADS

Recall that the serial port is used by rrload for UI interaction and it serves double duty when it is also used during an image download to the target platform. Take the following command for example:

```
rrload> copy -c kernel -s serial -d ram -f rrbin
```

This command is issued to rrload by using a terminal session running on a remote host connected over a serial port. Yet, the command itself is requesting a file transfer over that same serial line. Does the UI occurring over the serial line conflict with a file transfer over the same line? No it does not. When using the UI to request a file download, rrload drops into a mode where it stops using the serial channel for anything except bringing in the file image. Only when rrload brings in the final byte will rrload again start using the serial line for UI operations.

Now it is also true that when rrload drops into “load” mode, that the user must insure that only file image bytes are transferred over the serial line since any bytes received by the target device are interpreted as file bytes. It is important that the user is aware of this. In the case of using ‘minicom’ as a terminal window, we find that we can issue the “copy” command and then, without typing anything else in the terminal session, go to a shell window and type the following:

```
$ cat linux.rr > /dev/ttyS0
```

The above command streams the file to rrload. When rrload completes the file transfer, the user can move the cursor back over to the minicom window and resume interacting with the rrload UI.

ETHERNET DOWNLOADS

If you are planning to use the Ethernet port for your downloads then you must use the UI to establish parameter settings that allow the internal TFTP logic to work with your remote server. Option 3 from the main menu shows a list of current user settings. You can view these same settings from the command line mode by issuing the “set” command. The following are some example settings:

```
ui mode [cmdmenu]        :   menu
default boot cmd         :   boot_auto
I/O load format          :   rrbin
I/O load port            :   ether
Server IP       (tftp)   :   192.168.1.15
Server MAC      (tftp)   :   55:3:00:c0:00:F0
Device IP       (tftp)   :   192.168.1.50
Device MAC      (tftp)   :   11:22:33:EA:cc:00
Kernel Path     (tftp)   :   linux.rr
FileSys Path    (tftp)   :   romdisk.img.rr
Kernel cmdline            :
```

Note: The network stack implemented within the bootloader does **NOT** yet support the Address Resolution Protocol (ARP), therefore it is necessary to manually inform the remote server of your device IP/MAC mapping.

Normally the server would be able to use ARP at runtime to learn this mapping, however, until it can, use the command below on your server (**root** user may be necessary):

```
$ arp -s 192.168.1.29 00:e0:81:10:36:cf
```

Notice here that we are manually loading the server's internal ARP cache with the IP and MAC of our EVM device. It is important that your system administrator enables TFTP on your remote server, which is often disabled for security reasons. One final note, based on the above example, rrload will expect to find a kernel file by the name of **linux.rr** that should be located on the remote server in the default TFTP directory of that machine—in this case it is /home/<username>.

Passing a command line and Ethernet MAC to the kernel

The DSPLinux bootloader has the ability to pass a user supplied **kernel command line string** to the DSPLinux kernel as control is being transferred to it. Once the kernel boot process is under way, special logic in the DSPLinux kernel picks up the *command line* previously deposited by rrload then replaces the original default kernel command line embedded within the kernel. Using the normal method, the command line is then parsed in during the remaining steps of the kernel boot process. To implement this, a special SDRAM memory location is agreed upon in advance that, when built, both rrload and the kernel know about. Again, this allows the bootloader to deposit a command line string at that specific SDRAM memory location and when the DSPLinux kernel is booted, the string is picked up at the same location as just described.

A NULL terminated ***magic*** pattern string equal to "kcmdline-->" proceeds the real kernel command line string (if any) letting the kernel know that a valid string really exists (even if just a NULL string); if the pattern is found, then the kernel can assume that the command line string will immediately follow the magic string.

Both the magic string and the command line string (if they exist) butt up against each other, and again, are both null terminated strings; if the pattern is not found, then the kernel assumes that it was invoked by some other means than the rrload bootloader and avoids attempting to pick up an incoming command line string. Besides, there is probably nothing that is valid in there to pick up.

In like manner, the Device MAC rrload parameter configured by the user is also deposited in a special RAM location so that the kernel can retrieve it and make use of it in ways necessary for kernel Ethernet initialization. This NULL terminated Ethernet MAC string (for example, "00:11:22:33:44:55") is deposited in the same manner as described for the **kernel command line** and is preceded immediately by the NULL terminated "etherMAC-->" ***magic pattern***. When the kernel boots it can use these deposited strings as it wants.

Although the user does not usually need to know the exact RAM addresses used for depositing the **kernel command line** and **device MAC**, this information may help those who are debugging kernel boot issues. The specific RAM addresses used for the kernel's incoming command line and ether MAC are provided in Table 5 below.

Note: The quote (" ") characters are shown here for clarity only, with the NULL label representing the byte value zero (0) that terminates each string.

Table 5. RAM addresses for the kernel's incoming command line and ether MAC

Platform	Start Address	Comment
C547x EVM	`0xffc00020`	`"kcmdline-->"NULL"Command Line string"NULL`
	`0xffc00100`	`"etherMAC-->"NULL"00:11:22:33:44:55"NULL`

Note: The bootloader will manage these addresses for you based on the settings you established for the **kcmdline parameter** and **devmac parameter**. You can set these parameters by using the rrload menu mode interface (option 3 from the main menu) or the command line mode. Below is an example:

```
rrload> set kcmdline root=/dev/rom0
rrload> set devmac 00:11:22:33:44:55
```

BUILDING THE RRLOAD BOOTLOADER

```
$ cd dsplinux/c5471/rrload
$ cat README
$ make
```

This operation produces the following three rrload bootloader images.

Table 6.

Image	Description
rrload	• This is **ELF** version of the bootloader, which can be downloaded to the board's SDRAM via a JTAG-based emulator. *Note:* Currently this is the only way to get the very first instance of the rrload bootloader installed on the board.
rrload.rr	• This is an **rrbin** format version of the rrload bootloader as produced by the Makefile using the **mkimage** utility. It only differs from the ELF version in format. • This rrbin format is very close to a raw binary format. • This is used by a user performing an upgrade of an existing rrload installation. Other areas of this document discuss the process more fully.
rrload.ForUpGrade.rr	• This is identical to the other two version of the bootloader except it is built to load to a different SDRAM address. • This is used by a user performing an upgrade of an existing rrload installation. Other areas of this document discuss the process more fully.

UPGRADING AN EXISTING RRLOAD INSTALLATION

Here we discuss the installation of the rrload bootloader program and not the installation of a kernel and/or file system usually managed by the installed bootloader.

The Makefile of the rrload project automatically creates two rrload rrbin images, which are identical to one another except they are built to run at different locations within the memory map. This allows both instances of rrload to reside in memory simultaneously without conflicting with each other, which is necessary for performing rrload upgrades.

- The main instance is **rrload.rr** and represents the image you want loaded on the board.
- The second instance is called **rrload.ForUpGrade**. The purpose of the rrload.ForUpGrade object is to help provide the rrload user with the ability

to upgrade an existing rrload installation without the use of JTAG. The reason this is useful is to allow the following steps to occur for the user who wishes to upgrade an *old* instance of rrload with a *newer* version (again, without using JTAG).

Assuming the user has built the new version of rrload and has the current (old) running rrload in the command line mode, these steps will accomplish the upgrade.

Please note that in the example below, we are choosing to use Ethernet to load the new image and therefore we assume that you have the necessary bootloader parameters established to allow for this (see the main menu, Option 3). However, we also provide an upgrade example via serial following this first example.

Upgrading via Ethernet

1. Start a minicom window on the desktop connected to the target's serial port, then hold down the **ENTER** key while applying power to the board. Release the key after a second or two when you see the rrload main menu appear.
   a. At the main menu, select option 3.
   b. Make a note of any installation specific settings that you want to re-establish later after the upgrade process.
2. At the main menu, select option 5. This puts you in the command line mode where you type the following commands:

```
rrload> set kpath rrload.ForUpGrade.rr
rrload> copy -c kernel -s ether -d ram -f rrbin
rrload> eraseflash params
rrload> boot
```

Note: We used the "load kernel" path to sneak in the second instance of rrload. Then we booted it. At this point, the second instance of rrload should be running, we will use it to perform the upgrade. The RAM location previously used for rrload is now available so we can now download the new version of rrload to fill that spot. However, please note, you need to re-establish your desired bootloader parameters for the new image just loaded and running.

Please **DO NOT** store these settings to flash just yet. We still need to get the final bootloader image into memory, which is the next step.

3. At the main menu, select option 5 (command line mode), then type the following commands.

```
rrload> set kpath rrload.rr
rrload> copy -c kernel -s ether -d ram -f rrbin
rrload> boot
```

Note: The new version is in memory and is running. The only thing left to do is to establish your final bootloader parameter values (see Item 1-b) and then use the normal rrload commands to flash itself persistently as well as flash the parameters persistently.

4. At the main menu, select option 5 to get into command line mode, Then do the following:

```
rrload> eraseflash bootldr
rrload> copy -c bootldr -s ram -d flash -f na
rrload> set bootcmd boot_auto
rrload> set loadfmt rrbin
rrload> set loadport ether
        etc., for other desired settings, then...
rrload> copy -c params -s ram -d flash -f na
```

5. **FINISHED!** The rrload bootloader is upgraded and is stored persistently with its new bootloader parameter settings.

Note: Although this example assumed rrload command line mode, it could just as well have been performed using rrload's menu based UI.

Although this example performed the image "loads" via ether, it could just as well have been performed using one of the other I/O ports.

Upgrading via Serial

1. Start a minicom window on the desktop connected to the target's serial port, then hold down the **ENTER** key while applying power to the board. Release the key after a second or two when you see the rrload main menu appear.

   a. At the main menu, select option 3.

   b. Make a note of any installation specific settings that you want to re-establish later after the upgrade process.

2. At the main menu, select option 5. This puts you in the command line mode where you enter the following command.

   ```
   rrload> copy -c kernel -s serial -d ram -f rrbin
   ```

   Immediately do the following...

   a. On the host system, enter the commands below (where **ttyS1** is the serial port connected to the EVM):

      ```
      linux$ cd dsplinux/c5471/rrload/
      linux$ cat rrload.ForUpGrade.rr > /dev/ttyS1
      ```

   b. When the download completes, proceed with the following:

      ```
      rrload> eraseflash params
      rrload> boot
      ```

Note: We have used the “load kernel” path to sneak in the second instance of rrload. Then we booted it.

At this point, the second instance of rrload should be running, which we will use to perform the upgrade. The RAM location previously used for rrload is now available and we can download the new version of rrload to fill that spot.

3. At the main menu, select option 5 (command line mode), then enter the following command.

   ```
   rrload> copy -c kernel -s serial -d ram -f rrbin
   ```

   Immediately do the following...

   a. On the host system, enter the commands below (where ttyS1 is the serial port connected to the EVM):

      ```
      linux$ cd dsplinux/c5471/rrload/
      linux$ cat rrload.rr > /dev/ttyS1
      ```

   b. When the download completes, proceed with the following:

      ```
      rrload> boot
      ```

Note: The new version is in memory and is running. The only thing left to do is to establish your final bootloader parameter values (see Item 1-b) and then use the normal rrload commands to flash itself persistently as well as flash the params persistently.

4. At the main menu, select option 5 (command line mode) then enter the commands below.

```
rrload> eraseflash bootldr
rrload> copy -c bootldr -s ram -d flash -f na
rrload> set bootcmd boot_auto
rrload> set loadfmt rrbin
rrload> set loadport ether
        etc., for other desired settings, then...
rrload> copy -c params -s ram -d flash -f na
```

5. **Finished!** The rrload bootloader is upgraded and is stored persistently with its new bootloader parameter settings.

INSTALLING RRLOAD INTO A NEW FLASH CHIP

Getting the first instance of the bootloader onto the EVM, requires using the board's JTAG interface because no code is running on the board yet to "pull" in that first instance; JTAG allows you to "push" it there. Once rrload is loaded, use the rrload UI to request that it store itself in the on-board flash. With rrload present, you can load other software components such as kernel and file system through various standard board I/O ports (rather than JTAG loading).

The issue of upgrading rrload with a new version of rrload can be performed without the use of JTAG, this is described in "Upgrading an existing rrload installation" on page 98.

Standard JTAG

1. Install the jumpers on JP16, JP17, and JP18.

2. Next, use the emulator to download and run the setup program (built as part of rrload). This initializes the hardware making SDRAM visible.

3. Stop the emulator after a few seconds of running and then download and run rrload.

4. At this point, you can use the rrload UI to erase and program the flash.

Parallel JTAG

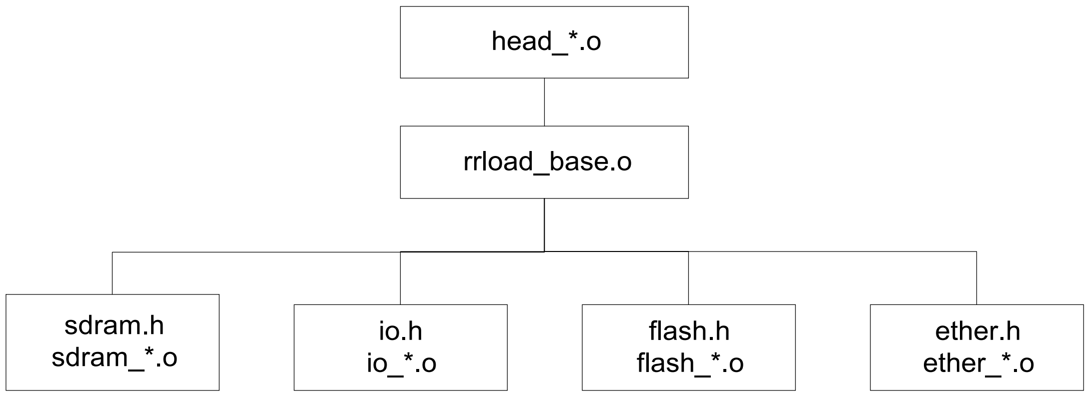
1. Remove the jumpers from JP16, JP17, and JP18.

2. Consult the GDB documentation supplied with your DSPLinux distribution and read the example session using the **sdi-arm-gdb debugger**. This shows how to install the bootloader, using the GDB debugger with the **sdserver** and the **parallel jtag port**. (See Chapter 4, section "GPP rrload/kernel debugger (sdi-arm-gdb)" on page 27.)

CUSTOMIZING RRLOAD WITH NEW DRIVERS

A Makefile links the following tree into a runable bootloader image called **rrload**. The individual ***.o** files were pre-built at the RidgeRun factory and are included in the rrload project directory along with the Makefile and the "index.html" document.

Figure 23.



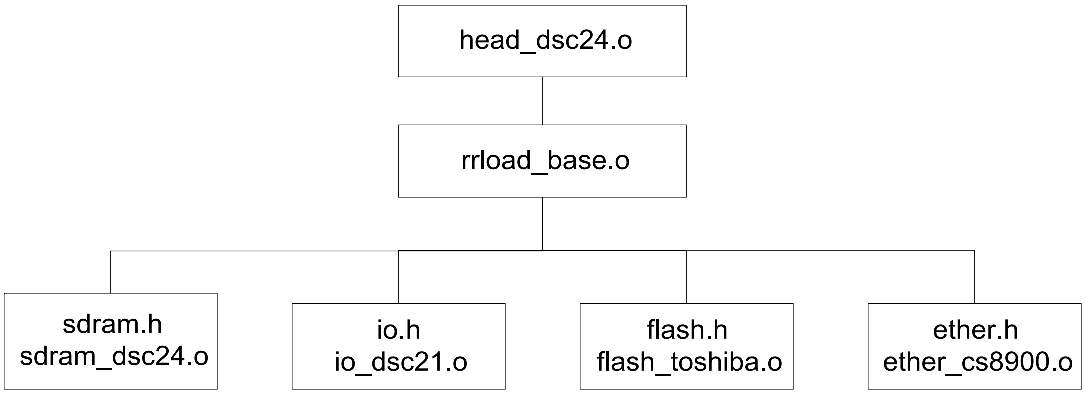
There is a ***.h** file associated with driver modules shown at the bottom of this hierarchy, which describe the interface of each driver. This allows a user to replace a driver with a new implementation. The new implementation must provide logic based on the header file. The user need only rename the existing ***.o** to some other name such as *.o.original, place the new ***.c** or ***.S** file in the directory, and then type **Make**. The Makefile compiles the new driver source file and then links it as normal, forming a *new* rrload image containing the replacement driver.

Important Note:
The examples below are taken from TI's DSC24 platform, not the C5471. These are supplied only as samples and do not apply directly to the C5471 platform.

Example 1: Replacing sdram_dsc24.o file with new version

In this example, we will replace the current **sdram_dsc24.o** file with a new version that supports the **4Mx32 Mobile SDRAM** instead of the SDRAM that typically ships with a DSC24 EVM. For starters, since we are using the DSC24 in this example, the specific directory structure, as shipped from the RidgeRun factory, looks like the diagram shown in Figure 24 below.

Figure 24. Default directory structure for DSC24 EVM



Replacing the sdram_dsc24.o driver

Follow the steps below to replace the current **sdram_dsc24.o** driver.

1. Rename existing driver object.

```
$ d rrload
$ mv sdram_dsc24.o sdram_dsc24.o.orginal
```

2. Bring in new driver impl. This is implemented as per the **sdram.h** description.

```
$ cp sdram_dsc24.S.example sdram_dsc24.S
```

3. Rebuild the bootloader (rrload).

```
$ make
```

Example 2: Source file

The following is the user supplied **sdram_dsc24.S.example** source file.

```
$ cat sdram_dsc24.S.example
.text
.align
.global sdram_ini
sdram_ini:
// SDMODE Reg (0x0x000309a6)
// =========================
 //   Bit(s) 15:     RDL select = “1 cycle”
 //   Bit(s) 14:     SDRAM bus width select = “32bits”
 //   Bit(s) 13:     DMA select = “ext mem -> SDRAM”
 //   Bit(s) 12:     Bank Number selection = “4 BANK”
 //   Bit(s) 11-10: Cas Latency selection = “3 cycle”
 //   Bit(s) 9-8:    Memory Type = “8k X 512 word”
 //   Bit(s) 7:      DQMC Control = “Normal”
 //   Bit(s) 6:      Automatic Power Down = “OFF”
 //
 // REFCTL Reg (0x0x000309a8)
 // =========================
 //   Bit(s) 8: Auto Refresh Enable = “Enabled”
 //   Bit(s) 7-0: Refresh cycle cnt = “0x10”
 //
 ldr r0,=0x000309a6
 ldr r1,=0x0000df00
 str r1,[r0]
 ldr r0,=0x000309a8
 ldr r1,=0x00000110
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df02
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
 str r1,[r0]
 ldr r0,=0x000309a6
 ldr r1,=0x0000df04
```

```
str r1,[r0]
ldr r0,=0x000309a6
ldr r1,=0x0000df04
str r1,[r0]
ldr r0,=0x000309a6
ldr r1,=0x0000df04
str r1,[r0]
ldr r0,=0x000309a6
ldr r1,=0x0000df01
str r1,[r0]
mov pc, lr // return to caller.
```

CUSTOMIZING THE FLASH PARTITIONS

The rrload bootloader sets aside four areas of flash that it uses for storing the 4-components it is designed to manage:

- The bootloader image
- The bootloader params
- The kernel image
- The file system image

Each of these components is given a very specific range of flash addresses that are involved whenever a user erases or stores that component. These four ranges are fixed at the time that the rrload Bootloader is built and are established in such a way that they fit the needs of most users. However, some users will want to re-define the way the Bootloader uses flash. Therefore, the Bootloader's design allows for a certain amount of re-configuring. Specifically, there are two control points we wanted to provide:

- Change the amount of space set aside for the kernel.
- Change the amount of space set aside for the file system.

The default flash memory ranges used by rrload for the various platforms are shown in the table below.

Table 7. Default flash memory ranges used by rrload

Platform	Kernel image	File system image
C5471 EVM	0x00040000 - 0x0011ffff	0x00120000 - 0x007fffff

To facilitate changing this partitioning, edit the rrload linker script and then re-build the rrload bootloader to get a version that reflects the new partitioning of kernel and file system space.

Example 3: Reconfiguring rrload's reserved space

Suppose you have the C5471 EVM and you have written some type of Linux driver that uses the last 0x20000 of the board's flash range. In this case you do not want the rrload bootloader to touch that portion of flash. However, you know that by default the rrload bootloader sets aside the whole flash range for the file system, extending from flash address 0x00200000 all the way to the end of flash at 0x007fffff. Moreover, you know that even if you only use a fraction of that space to store your Linux file system the whole area is essentially off limits to anything else since any rrload erase operation requested for the file system area will take out the whole range, not just the small part actually used at the moment.

For this scenario, you need to re-configure the bootloader so that it treats the file system storage as extending from 0x00200000 to 0x007dffff. Leaving the remaining 0x20000 bytes untouched by the rrload file system store/erase operations.

To accomplish this you need to **edit** the C5471 linker script:

```
dsplinux/c5471/rrload/ld.c5471.script
```

There are two lines in that file that we can change in order to re-partition the flash viewed by rrload. The two lines are

```
FlashFileSysStart = 0x0;
FlashFileSysEnd = 0x0;
```

Documentation inside that file describes the use of both of these. In our example here, all we need to do is **change** the one line *from this*:

```
FlashFileSysEnd = 0x0;
```

To *the line below,* then **rebuild** the bootloader.

```
FlashFileSysEnd = 0x007dffff;
```

For reference, the snippet of documentation from that file is reproduced below:

Example 4: Doc snippet from ld.c5471.script

```
When a user requests that rrload erase the
file system component, how does it know what range of
flash to actually erase? The answer is that the
internal source code defines the flash memory range
that each of the components will occupy (components
such as the bootloader itself, the bootloader
params, the kernel, and the file system).

The file system storage always follows the range set
aside for the kernel image storage which means that
if the kernel you store is significantly less than
the space set aside for it you will have a large pad
between the actual stored kernel image and the start
of the file system image. And likewise, if you have
kernel that is too large for the space set aside
then you will collide with the space set aside for
the file system. So...

—The Controls Offered—
The rrload linker script offers the user the ability
to override the internal location of the file system
and move the range of addresses that are set aside to
store it. This then becomes the new default for all
file systems downloaded and stored to flash and
becomes the new default used when the user requests
rrload to erase the file system range. The two
controls offered to the user are FlashFileSysStart
and FlashFileSysEnd. Each can be adjusted
independently and if either is set to 0x0 then this
turns off the override for that control and allows
```

Example 4: Doc snippet from ld.c5471.script (Continued)

the system to resort to the internal factory default
for that value.

—FlashFileSysStart—

This setting influences the size of the
file system storage area as well as the kernel
storage area. If you make the FlashFileSysStart
address lower than the internal factory default, then
you will be increasing the file system storage area
and decreasing the kernel storage area that proceeds
it. And likewise if you increase the FlashFileSysStart
address beyond the internal factory default then you
will be decreasing the file system storage area and
increasing the kernel storage area.

—FlashFileSysEnd—

If you redefine the FlashFileSysEnd address then you
can control the high address of the flash range
involved in the storage set aside for the file system
and likewise sets the end of the range effected by a
user request to erase the flash component. Changing
this number has no effect on the amount of storage
set aside for the kernel which proceeds the file system
storage area.

—Some Rules when Picking Values—

Legal values for FlashFileSysStart is any value that
begins on a flash block erase boundary. For
this platform that would be numbers such as
0x00040000, 0x00060000, 0x00080000, etc., since the
erase unit size of our flash is 128Kbytes (2*64; two
chips interleaved).

Example 4: Doc snippet from ld.c5471.script (Continued)

Legal values for FlashFileSysStart is any value that
ends on a flash chip erase block boundary. For this
platform that would be numbers such as 0x00003fff,
0x00005fff, 0x00007fff, etc., since

—Ramifications to your XIP Kernel/FS Builds—

Do not forget that the range of address set aside
for storing file system images has a header reserved
at the start of the range used for rrload meta-data
meaning that the first address actually available
for the user bits of a file system image are
FlashFileSysStart+0x20 which is the value that you
should assign during the DSPLinux kernel ‘make
xconfig’ session. See the FILE_SYS_START field on
the “general setup” panel of the ‘make xconfig’
session. This only applies however if you are
creating an XIP kernel/file system build since
otherwise the FILE_SYS_START will be set to a SDRAM
address and not a flash address and does not apply to
this discussion.

—Some Examples—

If FlashFileSysStart is set to 0x00080000 then the
earliest location I could use for my downloaded XIP
file system is 0x00080020. This would honor the
reserved space mentioned. This FlashFileSysStart
setting would allow the kernel images stored in
flash to extend to address 0x00080000-1. And for
XIP Kernel/FS builds, the following applies...
e.g. `mkimage --LAddr 00080020 romdisk.img
romdisk.img.rr which is used when the xconfig
session has 0x00080020 value defined for

Example 4: Doc snippet from ld.c5471.script (Continued)

```
FILE_SYS_START (General Setup Panel; XIP).

If FlashFileSysStart is set to 0x00120000 then the
earliest location I could use for my downloaded XIP
file system is 0x00120020. This would honor the
reserved space mentioned. This FlashFileSysStart
setting would allow the kernel images stored in
flash to extend to address 0x00120000-1. And for
XIP Kernel/FS builds, the following applies...
e.g. `mkimage --LAddr 00120020 romdisk.img
romdisk.img.rr which is used when the xconfig
session has 0x00120020 value defined for
FILE_SYS_START (General Setup Panel; XIP)
```

CHAPTER 12 Introduction to the DSPLinux Appliance Simulator

This chapter introduces the DSPLinux Appliance Simulator, describes a sample appliance, and gives detailed instructions on how to develop and debug programs in the simulation environment.

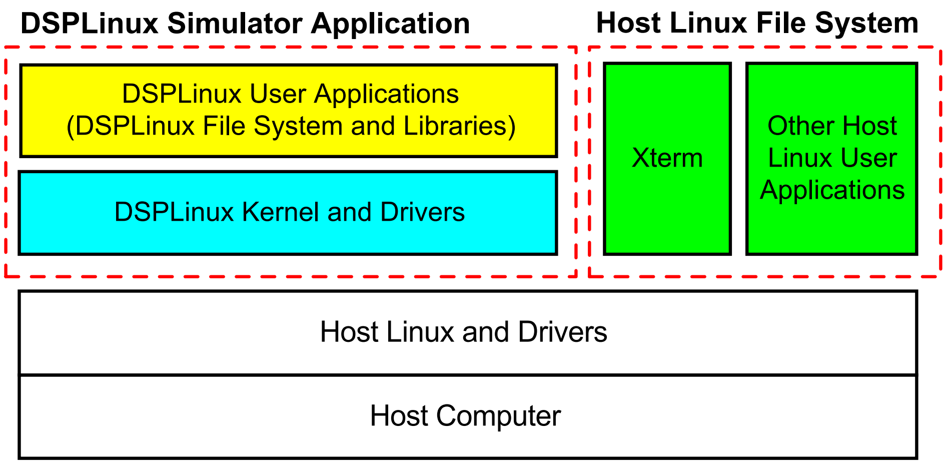
The simulator tool gives you the ability to develop, test, and implement your product while having limited access to hardware.

HOW SIMULATOR INTERACTS WITH THE HOST LINUX

The DSPLinux Appliance Simulator runs as an application on the "host Linux operating system." As shown in Figure 25 below, the DSPLinux Appliance Simulator application runs alongside other applications on the Linux host.

The Simulator implements a virtual machine running DSPLinux; it has its own file system and libraries, and runs applications within the virtual machine environment.

Figure 25. DSPLinux Appliance Simulator's relationship to the host computer


DSPLinux Simulator Application
Host Linux File System
DSPLinux User Applications (DSPLinux File System and Libraries)
DSPLinux Kernel and Drivers
Xterm
Other Host Linux User Applications
Host Linux and Drivers
Host Computer

File systems in the simulation environment

A file system is the collection of files used by the operating system. The file system includes the operating system, utilities, libraries, configuration files, and applications. For typical use of your host Linux operating system, all the files you interact with are on the host file system (for example, directories such as /etc, /dev, or /usr). The host file system is such an integral part of the operating system that, as a rule, it is not explicitly called out.

It is important to know about file systems when using the DSPLinux Appliance Simulator, because two different file systems are used.

- The first file system is the **host** file system that you work with while running desktop Linux on your host computer.
- The second file system is the **simulator** file system (see Figure 25 on page 113), which is the set of files used by DSPLinux when it is running in the simulation environment.

  When you actually build your embedded device that runs DSPLinux, you will have a small file system as part of the product. This second file system is stored in ROM or flash memory.

What you can do with the DSPLinux Appliance Simulator

Using the DSPLinux Appliance Simulator, you can:

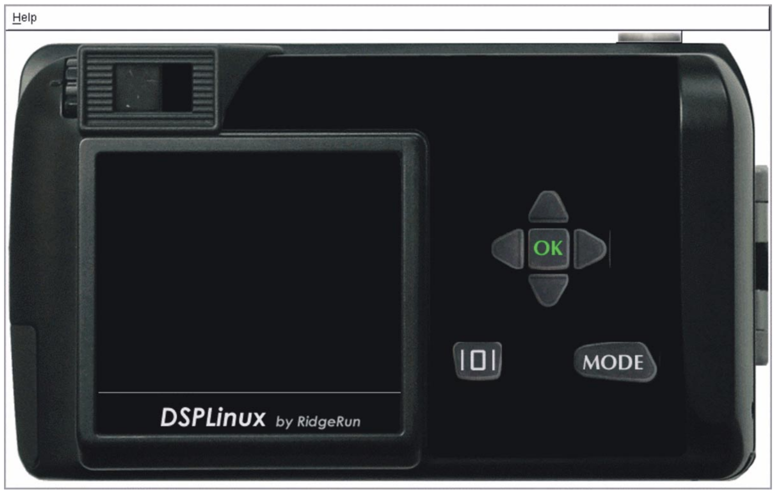
- Create and port applications to DSPLinux.
- Prototype the overall user-interface of your device.
- Define the interaction between applications, including those that need to communicate between the General Purpose Processor (GPP) and Digital Signal Processor (DSP).
- Determine the components of DSPLinux that you need in your system. (The mix of components affects how much ROM and RAM your product will need.)
- If you are making a device that will connect to a local network or the Internet, you can develop the networking applications in the simulation environment.
- Use cross-compile tools to develop applications for the simulated target. The same cross-compile concept is used to create code for the real target.

Note: The DSPLinux Appliance Simulator does NOT attempt to emulate the instruction set of an actual embedded processor (GPP or DSP). All code that executes on the simulated target is native x86 code.

WORKING WITH A SAMPLE APPLIANCE

To help introduce you to the DSPLinux Appliance Simulator, a sample simulated target—a camera—is shown below in Figure 26. The camera has a 640 x 320 LCD and control buttons.

Figure 26. Camera appliance sample



Starting the DSPLinux Appliance Simulator

Now that you know how the DSPLinux Appliance Simulator works, just follow the steps below to start the simulator and try out the sample camera.

1. From the host shell, type the following commands.

```
$ cd <development directory>/dsplinux/c5471/simulator
$ . ./setenv
$ ./camera
```

The simulator opens an **xterm** window on your host, representing the system console of the simulation environment. The standard Linux boot messages appear on the console window.

2. When the DSPLinux Appliance Simulator is fully booted, the message below appears.

   ```
   Welcome to the DSPLinux ARM7 distro by RidgeRun, Inc.

   Copyright (C) 2001 RidgeRun, Inc.
   Please report all bugs and problems to <support@dsplinux.net>
   ```

   – or –

   If the message does not appear on the console window, press ENTER.

3. Logging into the DSPLinux Appliance Simulator is just like logging into any other Linux machine.

4. At the login, enter the username **root**.

Note: The term **simulator shell** refers to the shell running on the simulated target. The term **host shell** means your normal shell running on your host computer.

5. The sample camera should pop-up on your host. The control buttons are not connected to any application. However, you can simulate pressing them by positioning your mouse pointer over one and clicking the left button.

Stopping the DSPLinux Appliance Simulator

Just as with the host Linux, DSPLinux running in the simulation environment must be shutdown when you are finished. From the simulator shell, follow the steps below to stop and shutdown the simulator.

1. At the host shell prompt, press **<CTRL>-C**.

2. Close the simulator windows using the close box in the upper right-hand corner. (There are 3-windows to close; the console window, application window, and splash screen. However, some or all of them may have already closed.)

3. Next, type the following command.

   ```
   $ killall -9 linux
   ```

   This closes anything running in the background and stops the simulator.

Simulator hangs or crashes

If the simulation crashes or becomes unresponsive, you will need to **kill** the processes associated with the simulator. To do this, type the command below in an available host system shell.

```
$ killall -9 linux sim
```

CHAPTER 13 Building an Application for the Simulated Target

This chapter provides a brief tutorial that walks you through the processes of setting up and working with a sample application.

TUTORIAL—BUILDING, TESTING, AND DEBUGGING APPLICATIONS

Now that you know how to start and run the DSPLinux Appliance Simulator, this tutorial will step you through the process of *cross-building* an application, *testing* it in the simulation environment, and *debugging*.

Cross-building

Many embedded developers are familiar with cross-building, which is the process of building code on your host for executing on a target system. When using the DSPLinux Appliance Simulator, you will cross-build to the simulated target environment. These are the same steps used to cross-build to your real target hardware environment, while using the appropriate DSPLinux cross-development toolset for your architecture.

1. Configure your development environment

In order to cross-build, you need to configure your development environment to use the **DSPLinux cross-build toolchain**. We will show you how to do this on the command-line and, of course, you can put this line in a shell script.

a. To configure the cross-build tools to use the DSPLinux Appliance Simulator as the target, enter the commands below in your host shell.

```
$ cd <development directory>/dsplinux/c5471/simulator
$ . ./setenv
```

The **setenv** script tells the DSPLinux example **makefiles** and **configure** scripts which DSPLinux target you are cross-building. When you are ready to go to a real target, you will run a similar script that selects the ARM cross-build environment.

Testing and debugging

This tutorial uses a sample application for testing and debugging. The application used is the hello application, which can be found at:

```
<development directory>/dsplinux/c5471/apps/hello
```

2. Building and adding an application to the file system

Let's begin by working with the sample **hello** application. In this example, we will build it and then add it to the target file system.

a. To build the application, type the following commands.

```
$ cd <development directory>/dsplinux/c5471/simulator
$ . ./setenv
$ cd apps/hello
$ make all
```

The compile should work with no errors or warnings, leaving the **hello** application executable in the current directory.

You may be tempted to test the application from your host shell, however, this executable will **NOT** run correctly because it was linked to run in the simulator.

b. Next, enter the commands below to add the application to the target file system.

```
$ make install
$ cd ../..
$ make fs
```

You have now successfully built and added the hello application to the target file system.

3. Debugging

You can debug applications running on the simulated target using remote debugging with **gdb**. The following example shows how to run **gdb** in the simulation environment.

Note: Before debugging, make sure that your code is compiled with the debug information (-g) option. (The **hello makefile** is already configured with the –**g** option.)

a. Type the commands below to launch the **gdbserver** on the simulated target.

```
# cd /usr/local/bin
# gdbserver :4321 /hello
```

The numbers **4321** specify which port to use. If your system is already using 4321, you can specify any available port number.

b. Next, enter the following commands, to run **gdb** on your host computer.

```
$ cd <development directory>/dsplinux/c5471/simulator/apps/hello
$ gdb hello
(gdb) target remote <simulator IP>:4321
(gdb) break main
(gdb) c
```

This assumes that you know how to use **gdb** to debug a program. The syntax and commands are the same as if you are debugging a program on your host computer. (For more information, see Chapter 4 “Using the GDB Debuggers” on page 17.)

Summary

This concludes the brief tutorial on how to use the DSPLinux BSP and Appliance Simulator to build, test, and debug your application.

To add your own applications to, or remove them from, the simulator file system, see “Simulator file system and application maintenance” on page 122. Once you have added some of your own applications into the file system, go to Chapter 14 “DSPLinux Toolchain and Appliance Simulator” on page 123 to review the more advanced techniques regarding the simulator.

SIMULATOR FILE SYSTEM AND APPLICATION MAINTENANCE

Adding applications

Use the sample makefile provided in "*<development directory>*/dsplinux/c5471/apps/hello" to see how to add applications to the file system.

```
$ cd <development directory>/dsplinux/c5471/apps/hello
$ . ./setenv
$ make clean all install
$ cd ../..
$ make fs
```

Removing applications

To free up space, enter the commands below to remove unwanted applications from the file system.

```
$ cd <development directory>/dsplinux/c5471/fs
$ . ./setenv
$ rm -rf TIDSC21_EVM/binaries/*
$ rm -rf X86SIMu/binaries/*
$ su -c 'rm-rf fs'
$ cd ..
$ make fs
```

CHAPTER 14 DSPLinux Toolchain and Appliance Simulator

This chapter contains information on cross-building the DSPLinux **Toolchain** and **Appliance Simulator**. In addition, guidelines are given for sharing libraries, cross-building using the **DSPLinux Make**, and for using the **PATH** environment variable.

For standard Linux tools and libraries, more documentation is available on the Internet. For some starting points, see Appendix D - "URL Links to Related Web Sites" on page 133.

.

Note: The tutorial in Chapter 13 introduced you to the basics of the DSPLinux toolchain and Appliance Simulator. Because there may be multiple ways to accomplish a task, the method used in the tutorial is called out when appropriate.

CROSS-BUILDING WITH TOOLCHAIN, LIBRARIES, AND INCLUDES

You must do two things to cross-build applications for the DSPLinux Appliance Simulator:

- Use the DSPLinux **toolchain**.
- Use the **libraries** and **include** files in the DSPLinux Appliance Simulator's development file system.

Note: In the development file system, the **libraries** and **include** files are already built. They were cross-built for the simulator target using the simulator toolchain.

Using the DSPLinux toolchain

For details on configuring your development environment and using the DSPLinux cross-build toolchain, see "Cross- building" on page 119.

Using the DSPLinux libraries and include files

It is important to include and link to the libraries that are part of the simulator's expanded file system to ensure that your package will run correctly in the simulator environment. To use these **libraries** and **includes**, it is useful to understand how the Appliance Simulator toolchain works.

- The DSPLinux cross-build compiler searches for **include files** in:
  - any paths passed into the compiler with a **-I** option
  - `/opt/DSPLinux/X86SIM/crossdev-uclibc/include`
- The cross-build linker searches for **libraries** in:
  - any paths passed into the linker with a **-L** option
  - `/opt/DSPLinux/X86SIM/crossdev-uclibc/lib`

Note: The compiler and linker will search in the paths passed in with -I and -L **BEFORE** looking in the standard path. Thus, passing in paths to the **includes** and **libraries** on your host workstation [using **-I** and **-L**] could result in the use of the **WRONG** includes and libraries.

CROSS-BUILDING WITH MAKE RULES

The preferred approach to cross-building is to use the DSPLinux **make** rules, which define build tool macros for your project. These rules read the DSPLINUX_ARCH environment variable (which is set by the **setenv** scripts), to determine which DSPLinux target you are building.

Note: If DSPLINUX_ARCH is **NOT** set, your project is built natively, using your host workstation's build tools.

Set your DSPLINUX_ARCH environment variable to DSPLinux target

```
$ cd <development directory>/dsplinux/c5471/simulator
$ . ./setenv
```

Using DSPLinux make rules

1. Modify your project's makefile to include the DSPLinux **make** macro definitions at the top of your **makefile**. (Be sure that you do not re-define these macros so that they no longer refer to the correct DSPLinux cross-build tool!)

   Below is an excerpt from "`/opt/DSPLinux/X86SIM/examples/hello/Makefile`":

```
#
# Include the common DSPLinux rules to set build tool macros
# (CC,LD,etc) based on DSPLINUX_ARCH environment variable. If
# DSPLINUX_ARCH is not set, tool macros will be defined with
# standard names (and found in your default PATH).
#
include /opt/DSPLinux/rules/makeinclude
```

2. To make use of the cross-build tool macros, run **make** on your modified makefile(s).

```
    $ make
```

CROSS-BUILDING USING THE PATH APPROACH

This approach is useful for cross-building existing packages containing **make-files** and **configure** scripts that are difficult to modify for use with the DSPLinux build rules.

Changing the Path environment variable

1. To change your **PATH** *environment variable* to use DSPLinux toolchain for the simulator, type the command shown below.

Note: Once you do this, all programs you compile while in this shell are then built using the DSPLinux toolchain. This could cause problems if you want to compile an application natively for your host computer.

```
$ export PATH=/opt/DSPLinux/X86SIM/crossdev-uclibc/i386-linux/bin:$PATH
```

CAUTION: Take care to **NOT** pass in **–I** or **–L** options to **includes** and **libraries** that were not cross-built for the correct target.

2. You can verify which compiler you are using with the **which** command below.

```
$ which gcc
/opt/DSPLinux/X86SIM/crossdev-uclibc/i386-linux/bin/gcc
```

3. Once you are finished cross-building, it is recommended that you exit your shell to ensure that you do not unintentionally continue to use the DSPLinux toolchain.

Other information

For more information, please refer to the following:

- the man and info pages for **make**
- `<`*`development directory`*`>/DSPLinux/c5471/apps/hello/Makefile`
- the files in the `/opt/DSPLinux/rules` directory

APPENDIX A Jumpers for the C5471 EVM

This appendix lists the jumper connections for the C5471 EVM jumpers.

Note: Before powering on the C5471 EVM, ensure that the jumpers are connected as indicated below. In addition, be sure to follow the configuration steps outlined in Chapter 2 “Turning on the C5471 EVM” on page 7.

Table 8. Jumper connections

Connection	Jumper
No jumper	JP4, JP8, JP9, JP10
Connect a jumper between pin 1 and pin 2 for:	JP1, JP2, JP7, JP13, JP14, JP15, JP20
Connect a jumper between pin 2 and pin 3 for:	JP3, JP5, JP6, JP19, JP21
Jumper installed	JP11, JP12, JP23, JP24 * If using a Standard JTAG connector, then: JP16, JP17, JP18 installed
Jumper removed	** If using a parallel JTAG connector, then: JP16, JP17, JP18 removed
Of the 6-pins, jumper the 2-pins that are closest to the Parallel port	JP22

APPENDIX B DSPLinux C5471 Driver Reference

The drivers that come with the C5471 BSP are listed in the table below.

Table 9. C5471 device drivers

Device Driver	Description
`/dev/timer0`	Timer0; \`insmod timer\`
`/dev/timer1`	Timer1; \`insmod timer\`
`/dev/watchdog`	Timer0; \`insmod timer\`
`/dev/spi0 -- /dev/spi2`	SPI; \`insmod spi\`
`/dev/gio0 -- /dev/gio35`	GPIO; \`insmod gio\`
`/dev/gio36 -- /dev/gio43`	LEDS; \`insmod gio\`
`/dev/gio44 -- /dev/gio51`	Switches;\`insmod gio\`
`/dev/ttyS0, /dev/console`	Serial Port
`/proc/pm/*`	Pwr Mgmt; ‘insmod pmck, insmod pmproc’

APPENDIX C Troubleshooting and Support

TECHNICAL SUPPORT

RidgeRun hosts the site http://www.dsplinux.net as a forum to provide updates and support to DSPLinux developers. To communicate with RidgeRun engineers and other users working with DSPLinux, you may want to subscribe to one of the developer mailing lists hosted at http://www.dsplinux.net.

As a user, we would appreciate your feedback, defect reports, and suggestions.

Known defects

If you encounter a problem, check the list of known defects in the "OpenDefects.pdf" file at the path given below. This contains the list of defects that we knew about at the time of this release. If additional defects become known they the list is updated and is posted to http://www.dsplinux.net.

```
<development directory>/DSPLinux/c5471/build/OpenDefects.pdf
```

APPENDIX D URL Links to Related Web Sites

Refer to the URLs below for additional information relevant to the DSPLinux C5471 BSP.

Linux documentation	For Linux documentation, including Guides and FAQs, go to: http://www.linuxdoc.org
Linux Newbie.org.	For helpful guides to new Linux users (newbies), go to: http://www.linuxnewbie.org
Linux kernel	For new versions of the Linux kernel and the Linux kernel archives, go to: http://www.kernel.org
Memory Technology Device (MTD)	For details on the development of the MTD Linux subsystem for memory devices, especially Flash devices, go to: http://www.linux-mtd.infradead.org/
uClinux	For information on uClinux, which is a derivative of the Linux 2.0 kernel intended for micro controllers without Memory Management Units (MMUs), go to: http://www.uclinux.org/
User-mode Linux	For brief descriptions of how people are using the User-mode Linux, got to: http://user-mode-linux.sourceforge.net

APPENDIX E Glossary

ARM Ltd	RidgeRun's DSPLinux operating system, is a version of Linux that includesPreviously named Advanced RISC Machines, ARM Ltd is the developer of the ARM embedded RISC microprocessor family.
ARP	Address resolution protocol.
Board support package	Tools, drivers, Linux kernel, and documentation necessary to support development on a particular board.
Bootloader	RidgeRun's proprietary code used to load code to flash and set boot options. See "rrload."
bpp	Bits per pixel.
BSP	*See* "Board support package."
Busybox	Open-source code that combines tiny versions of common utilities and a shell into a single small executable.
Cross-build	The process of running a toolchain on your host computer to produce executables for a different *target file system*, also known as "cross-compile." For example, you may cross-build on the host to produce files for the real target.
Daemon	A process that typically runs in the background, behind the scenes, that is usually invoked as part of the system startup scripts. These processes usually run continuously as helper utilities for various system operations such as networking.
DHCP	Dynamic Host Configuration Protocol server.
Digital Signal Processor	A microprocessor specialized to process data in real-time. Certain operations can be executed in parallel, which results in higher processing capacity for a given clock speed.
DSP	*See* "Digital Signal Processor."

DSPLinux	RidgeRun's DSPLinux operating system, is a version of Linux that includes programming tools, several packages for specific devices, and simulators for some designs.
DSPLinux Simulator	Refers to the DSPLinux operating system running as an application on the Linux host.
Evaluation module	The evaluation module (EVM) is the TI circuit board containing the C5471 chip.
EVM	*See* "Evaluation module."
File system	The directory structure (beginning with "/") comprising configuration files, applications, libraries, and an operating system.
General purpose processor	A microprocessor that is **not** specialized for real-time parallel processing of data.
GNOME	*See* "GNU Networked Object Model Environment."
GNU Networked Object Model Environment	A user-friendly set of applications and desktop tools used in conjunction with a window manager for the X Window System. GNOME is similar in purpose and scope to CDE and KDE, but GNOME is based completely on Open Source software.
GPP	*See* "General purpose processor."
Host computer	The Linux-based desktop system used for software development, and to run the DSPLinux Simulator.
Host Linux	Refers to the Linux operating system running on the *host computer*. *See also* "User-mode Linux."
Host shell	The Linux terminal shell running on the *host computer.* The default prompt is "`$`" for normal users and "`#`" for root.
IP	Interent Protocol.
JFFS	The Journaling Flash File System, developed by Axis Communications in Sweden, is aimed at providing a crash/powerdown-safe file system for disk-less embedded devices. It is released under the GPL, and the current version works for the Linux 2.0 kernel series and memory-mapped industry-standard flash-memories (aka "NOR-flashes").

JFFS2	A Journaling Flash File System based on the original JFFS and developed by Red Hat. JFFS2 offers improved performance, compression, RAM footprint, concurrency and support for suspending flash erases, and the support for hard links.
JTAG	Joint Test Action Group, specifies test framework for electronic logic components; IEEE Standard 1149.1, also known as IEEE 1149.1 (JTAG) or Boundary Scan.
KDE	*See* "K Desktop Environment."
K Desktop Environment	A powerful Open Source graphical desktop environment for Unix workstations.
Linux kernel	The core of the Linux operating system, including networking.
Minicom	Minicom is a menu-driven, serial communication program, and includes ANSI color, dialing directory, dial-a-list, script language, and so forth. Minicom is a clone of the SMDOS Telix program.
Real target	An actual device running DSPLinux.
rrload	The RidgeRun rrload bootloader allows users to manage the loading, storing, and invoking of a Linux kernel and root file system. In normal operation, the bootloader resides in flash and is the first program run when powering up. Unless intercepted, rrload will typically transfer control to the stored system.
SDK	Software development kit.
Simulated target	A software representation of a device that runs DSPLinux (a simulation of a real target). The Simulated Target displays a representation of the device in a window on the *host computer*, which shows the screen, lights, and buttons of the device.
Simulator shell	The Linux terminal shell running on the *simulated target*. The default prompt is "#".
SOC	System-on-a-chip.
Target file system	The file system for a particular target device.
TI	Texas Instruments, Inc.

Toolchain	Tools and utilities used to build applications and other code for the target.
UI	User interface.
User-mode Linux	Linux running as an application on a host Linux. The DSPLinux Simulator utilized user-mode-Linux to run DSPLinux as an application on host Linux.
Virtual Network Connection	Any connection between a simulated target and another entity on the network. It is common to have a virtual network connection between the simulated target and the host computer.
x86	Any processor compatible with the Intel 386, 486, or Pentium family.
XIP	Execute in place.

INDEX