Embedded Linux Made Easy

Microcross GX-Linux™

Enterprise Platform Edition
Embedded Linux Development Solution

User Guide

Version 2.0-20080212

© 2008 Microcross, Inc.
Licensed Materials

MICROCROSS PRESS

Powered by GNU X-Tools™, SlickEdit® & Linux OS

ISBN: N/A

Microcross, Inc.
202 Blossom Lane
Centerville, GA 31028

© 2008 Microcross, Inc.

All rights reserved.

This document and the GX-Linux™ Embedded Linux BSP described in it are furnished under license with Microcross, and as such, may be used or copied only in accordance with the terms of the license. The contents of this document are furnished for informational purposes only and are subject to change without notice. Microcross assumes no responsibility or liability for any errors or inaccuracies that may exist in this document. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, without the prior written consent of Microcross.

Trademarks

GX-Linux™, Visual X-Tools™ and GNU X-Tools**™**, *MICROCROSS PRESS*™ and Microcross**®** are all trademarks of Microcross, Inc.. All other brand and product names, services names, trademarks and copyrights are the property of their respective owners. This documentation contains copyright materials and has been prepared by Microcross Technical Publications; contact the Microcross Technical Publications staff for more information: **support@microcross.com**.

Disclaimer

Microcross, Inc. makes no representations or warranties with respect to the contents or use of this user guide, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Microcross, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. Microcross, Inc. makes no representations or warranties with respect to any Microcross software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Microcross, Inc. reserves the right to make changes to any and all parts of Microcross software, at any time, without any obligation to notify any person or entity of such changes.

GX-Linux™ User Guide
Supports Product Part Numbers: MGXL4xxLx-xxx, February 2008
Printed in USA

MICROCROSS PRESS

Table of Contents

Section 1.0 Overview and System Requirements 7

1.1 Overview 7

1.2 Components of the GX-Linux Enterprise Edition 7

Section 2.0 Installation and Setup 8

2.1 Enterprise Edition Installation and Setup on a Linux Host Workstation 9

2.2 Uninstall GX-Linux 10

Section 3.0 Getting Started 12

3.1 Getting Started with GX-Linux on a Linux Host 13

3.2 MicroMonitor Boot ROM Startlinux Familiarity and Tips 17
3.2.1 Manual TFTP Up-Load Procedure 17
3.2.2 Useful Startlinux Commands 18
3.2.3 Other Useful MicroMonitor Commands 19

3.3 Sound Example with SOX (if supported by the target board) 19

3.4 Telnet Usage with Target Board 20

Section 4.0 Visual X-Tools Example Application Project 21

4.1 Hello Example Program 21

Section 5.0 Making Modifications to GX-Linux 23

5.1 Modifying GX-Linux Enterprise Edition 23
5.1.1 GX-Linux BSP Components 23
5.1.2 Building All of the Components 23
5.1.3 Building All Components Excluding Examples 23
5.1.4 Building Your Own Applications 23

5.2 Changing the Kernel Configuration 24
5.2.1 Configuration Files 24
5.2.2 Changing the Working Configuration 24
5.2.3 Resetting the Kernel to the Default Configuration (disaster recovery) 25
5.2.4 Customizing the Kernel Configuration 25
5.2.5 Modifying Kernel Loader Address 26

5.3 Changing the Target File System 26
5.3.1 Adding Content to the Target File System 26

5.4 Putting the New System on the Target Hardware 26
5.4.1 Putting the New System on the Target Hardware 26

5.5 Read-Only and Read-Write File Systems 27
5.5.1 ROMFS File System 27
5.5.2 NFS Mounted Root Filesystem 27
5.5.3 Writing to On-Board Flash via a Pendrive (USB) – If Supported on Target 29

5.5.4 How to Prepare a Pendrive with a Root Filesystem and Work with GX-Linux 29
5.5.5 Writing to On-Board Flash via MMC/SD Card – If Supported on Target 30
5.5.6 Writing to On-Board Flash via a Compact Flash Card – If Supported on Target 30

Section 6.0 Debugging Applications 32

6.1 The GDB Debugger 32
6.1.1 GDB Debugger 32
6.1.2 gdbserver 32
6.1.3 gdbserver Communications 32
6.1.4 Graphical Front-Ends 32
6.1.5 Debug-able Files 33
6.1.6 Visual GDB 33

6.2 Debugging Examples 33
6.2.1 Example with Visual GDB 33
6.2.2 Example with GDB 33

6.3 Debugging Using the .gdbinit 35

6.4 Debugging Linux Kernels and Drivers with the BDI2000 35

7.0 Networking 41

7.1 Default Kernel Configuration with Networking Enabled 41

7.2 Network Configuration 41

8.0 Serial Port 43

8.1 Serial Devices 43
8.1.1 Introduction 43
8.1.2 Default Serial Driver Configuration 43

8.2 Program Access to `/dev/ttyS0` 43

8.3 System Console 43
8.3.1 Default Console 43
8.3.2 Login via the Serial Port 43
8.3.3 Standard IO 43
8.3.4 Restrictions 44

8.4 Serial Settings 44

8.5 Kernel Console over Serial 44

Section 9.0 How to Use GNU X-Tools 46

9.1 Introduction 46

9.2 Using the GNU X-Tools Command Line Tool `(xtools)` 46

9.3 Invoking GNU X-Tools 47

9.4 Using the GNU X-Tools Toolsuite 48
9.4.1 Simple Example 48
9.4.2 GNU X-Tools Toolsuite Description 49
9.4.3 Control Program (GCC) 50

9.5 Controlling the Tools Using the GCC 53
9.5.1 GCC Options Commonly Used 53

9.6 Controlling Linker from GCC 55
9.7 Compilation Options 56
9.7.1 Displaying compiler behavior 56
9.7.2 C Language Options 56
9.7.3 Preprocessor Options 57
9.7.4 Options to Specify Libraries, Paths and Startup Files 57
9.7.5 Debugging and Profiling Options 59
9.7.6 Optimization 60
9.7.7 Passing Options to the Assembler or Linker 61
9.8 Using the GNU Assembler 61
9.9 Using the Linker 64
9.9.1 Invoking ld 64
9.9.2 Linker Options 65
9.9.3 Linker Scripts 66
9.9.4 Link-Order Optimization 67
9.9.5 The C Runtime (crt0) 67
9.10 Object Translation (ELF to Binary, SREC, etc.) 68
9.11 Creating/Updating Libraries 68
9.12 GNU Libraries 69

Appendix 1. GNU X-Tools ARM Linux Toolsuite 70

Appendix 2. Common Minicom Problems 78

Appendix 3. Glossary 79

Appendix 4. Additional Resources 81

Section 1.0 Overview and System Requirements

1.1 Overview

The GX-Linux User Guide for building Embedded Linux applications has important tutorial information on how to use the BSP and tools effectively and jump start the learning process so you can focus on developing your application. Specifically, this User Guide provides information on how to get the GX-Linux setup with GNU X-Tools and GX-Linux Board Support Package (BSP) and building applications for the target board. These are the requirements.

System Requirements for GX-Linux Software BSP Installation

1. Hardware:
   Linux OS and Host with an x86 or Intel® Pentium® family processor
   keyboard
   mouse,
   video monitor

2. Linux host OS distribution pre-installed on an x86 computer:
   Red Hat® Linux 9.0 or newer (complete install)
   Debian (complete install)
   Linspire® Linux Developer's Edition

3. CPU clock rate: 400 MHz or higher x86 PC

4. Recommended system RAM: 256 MB

5. Hard disk space required: 1 GB

6. A network card installed and configured on the host computer

7. RS-232 serial port (female-to-female, 9-pin, NULL modem cross-over or a hub for Ethernet connection)

8. Ethernet cross-over cable for standalone configuration or two standard UTP patch cables for a LAN / Workgroup configuration.

1.2 Components of the GX-Linux Enterprise Edition

The GX-Linux Enterprise Edition package includes a single compact disk (CD-ROM) loaded with GNU X-Tools, documentation, GX-Linux BSP source and example programs, plus a second CD-ROM with Visual X-Tools IDE. Go through the installation and initial setup procedures in the next section, and then go to Section 3 to get started.

- Cogent uSBC OEM board and breakout board with power supply, null modem serial cable, and cross-over Ethernet cable
- Linux kernel v2.6.x sources for the target processor
- glibc and stdlibc++ libraries (pre-built) – a full C/C++ library for Embedded Linux users on ARM/XScale platforms
- Busybox on a ROM filesystem with basic system utilities and example applications
- GX-Linux Master Makefile for building everything in the BSP
- Bootloader utility for starting Linux on MicroMonitor boot ROM

Section 2.0 Installation and Setup

This section covers a number of procedures to install all components of the complete Microcross GX-Linux solution.
Note: GX-Linux kernel can only be built on a Linux Host Platform.

Section 2.1
Enterprise Edition Installation and Setup on a Linux Host Workstation

Section 2.2
Uninstall GX-Linux

2.1 Enterprise Edition Installation and Setup on a Linux Host Workstation

The installation procedure for the Enterprise Edition is a simple command line procedure. We assume that all requirements in Section 1 are met and that you have a Linux host OS running on your workstation or server before getting started with this installation. All commands are done from the Bash/xterm shell in Linux.

Step1.

Insert the GX-Linux distribution CD into your host computer's CD-ROM drive.

Step 2.

Start a Bash/xterm shell and login as `root` (or use `su`).

Step 3.

Mount the GX-Linux distribution CD on the CD-ROM drive using the `mount` command,

Example:

```
# mount -t iso9660 /dev/cdrom /mnt/cdrom
# exit
```

The system will announce that it has mounted the CD-ROM as read-only. **Note:** On some Linux systems, the CD-ROM will be mounted by an automounter, which means using mount in step 3 is not required, or the procedure may need to be modified to accommodate non-standard Linux distributions or a machine with multiple CD-ROM drives that have differing naming conventions.

Step 4.

Select or create an *install* directory on your Linux host machine, preferably in your `/home/<user-name>` directory or a directory where you have full permissions. We call this the *install* directory because the *build* directory will be untarred to this directory. The *build* directory is named `gxlinux-<ed> (either gxlinux-ent or gxlinux-pro)`, and when you untar the GX-Linux BSP tarfile, it creates this build directory at the point of install. The Build contents for GX-Linux are shown in Table 2.2.

Table 2.2 Enterprise Platform Build Directory

Directory / File Name	Description of gxlinux Enterprise Platform Build directory
`addons`	Directory contains sound programs and Microwindows sources
`busybox`	Directory of BusyBox utilities
`config`	Default Linux target configuration files.
`examples`	Directory of 22 example applications that are built and copied into the rootfs filesystem (/usr/bin) and one Visual X-Tools example project.
`linux`	Directory of the kernel source tree.
`Makefile`	Master makefile
`rootfs`	Directory of root filesystem
`system.cfg`	File created at make time that stores configuration settings (BOARD and ARCH).
`varetc`	Directory contains the filesystem etc and var directory contents; the etc directory contains the system configuration files for the host, and the var directory contains logs, lock files and process specific data files.

Table 2.3 shows the *source* directory contents; there is only one file that is needed to install: `gxlinux-<ed><ver>-<board>-bin.tar.gz` and the next step shows how to install this tarfile.

Table 2.3 Source Directory (src) on CD-ROM

Source File Name	Description
`gxlinux-ent-<ver>-<board>-<date>.tar.gz`	GX-Linux Enterprise Edition BSP

Step 5.

Browse the CD-ROM src directory to get the complete filename (format is `gxlinux-<ed><ver>-<board>-<date>.tar.gz`).

Example with notional names: Installing GX-Linux BSP

- Open an xterm/Bash shell.
- Mount the GX-Linux CD-ROM if it is not already mounted.
- **Do Not** use su or root privileges for this next procedure. Issue the following commands as your user name to create a build directory:

```
$ cd /home/<user-name>  (other other install directory)
$ tar xvzf /mnt/cdrom/src/gxlinux-ent-v1.0m-csb535fs-20060710.tar.gz
```

Note: You might need to substitute the right path to the CD-ROM if your system differs from the standard CD-ROM mount pathing. The build directory gets created at the point you install the GX-Linux BSP and is named `gxlinux`.

Step 6.

Install your GNU X-Tools Linux Toolsuite per instructions in the README of the CDROM; then, continue here to set your permissions on the host Linux operating system so you can write to `/usr/<target-alias>` directory and have `su` privileges modifying library and header directories.

Example using the arm-linux toolsuite (substitute `<target-alias>` and `<user-name>` accordingly)

Open an xterm/Bash shell
Login as su or root

```
# chown  -R  <user-name>  /usr/arm-linux
# chmod  -R  775  /usr/arm-linux
```

With an editior like `vi`, edit as `root` user the `/etc/sudoers` file to add your user name to the super users list. An example follows. Enter two lines below the `sudoers` file's user alias specification (assuming a Red Hat Linux sudoers file);

```
Defaults:<user-name> timestamp_timeout=-1
<user-name> ALL=(ALL) ALL
```

Now save sudoers file and exit from root

```
# exit
```

At this point, the BSP and toolsuite are installed and permissions are set. Next, go to Section 3 to configure the target board, serial interface, network settings, and then run GX-Linux and example applications on the target single board computer.

2.2 Uninstall GX-Linux

Linux Host

To remove a previously installed version of the GX-Linux BSP software, issue the following command. WARNING: Before removing the tree, be sure to backup any files you want to keep.

```
$ cd /<peer-dir-to-gxlinux-<ed>>
$ rm -rf gxlinux-<ed>
```

Section 3.0 Getting Started

Below are the names of the appropriate sections for getting started with your Enterprise Edition.

Section 3.1
Getting Started with GX-Linux on a Linux Host

Section 3.2
MicroMonitor Boot ROM and Startlinux Familiarity and Tips

Section 3.3
Sound Example with SOX (if supported by the target board)

Section 3.4
Telnet Usage with Target Board

3.1 Getting Started with GX-Linux on a Linux Host

Board Connection and Initial GX-Linux Bootup
Make physical connections per the diagram layout in Figure 3.1. If connecting to a hub on a network, disregard connecting an Ethernet cross-over cable. To get the target board booted up and communicating with your host Linux computer, you will need to configure your serial port and MicroMonitor before loading GX-Linux binaries.

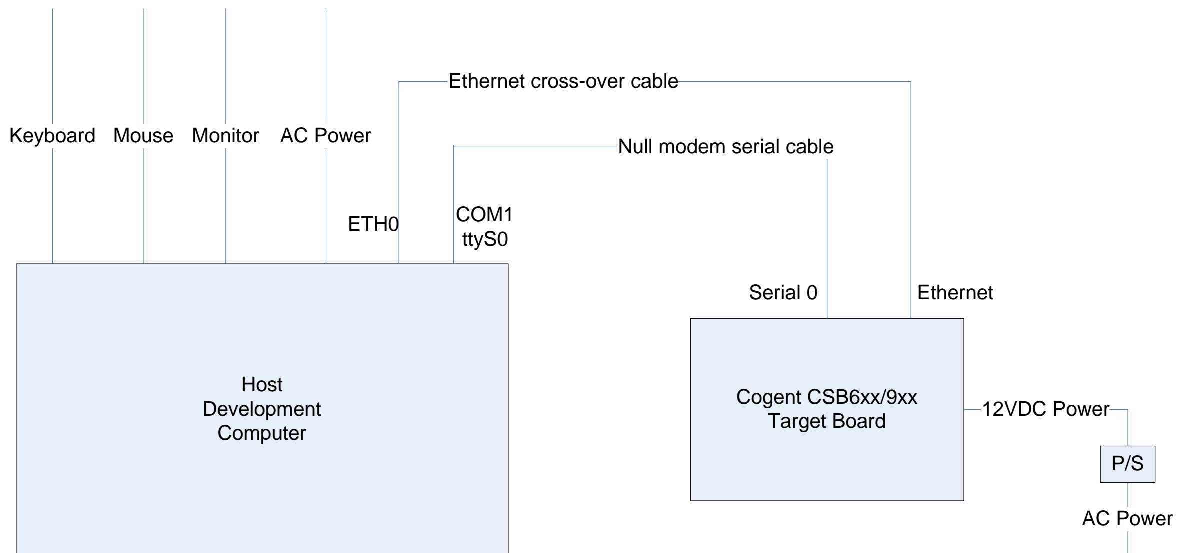
Step 1. Setup Serial Terminal

- If you do not have a serial terminal program running on your host machine, you will need to install one. We recommend a serial terminal program such as `minicom` or similar, which can be downloaded over the Internet.
- Connect the Cogent provided null-modem serial cable to the target board and host computer.
- Configure your serial terminal program to connect to the target board using the following settings:

```
38400 baud
8 data bits
No parity
1 stop bit
no flow control
```

- Connect the power supply to the board, and the serial terminal should show the MicroMonitor startup banner and the uMON> command prompt.

Figure 3.1 Diagram of Board Connections to Host Computer



Step 2. Setup uMON

You have two options to setup the network file, `monrc`, in MicroMonitor: manual settings and DHCP/BootP. Consult your system administrator if you need help in determining your network settings.

Manual network settings: The settings shown are notional and are for reference of the procedure only. Use the uMON command line interface to your serial terminal session to enter these commands.

```
uMON> set -c                          (clears monrc memory)
uMON> set IPADD 192.168.0.110         (IP address)
uMON> set NETMASK 255.255.255.0       (network mask)
uMON> set GIPADD 192.168.0.250        (gateway IP address)
uMON> set -f monrc                    (saves settings to monrc)
uMON> reset                           (reboots)
```

DHCP/BOOTP network settings: Follow this example and enter the commands using your serial terminal uMON command line interface.

```
uMON> set -c            (clears monrc memory)
uMON> set IPADD DHCP    (or DHCPv or DHCPV or BOOTP)
uMON> set -f monrc      (saves settings to monrc)
uMON> reset             (reboots)
```

Select only one for IPADD setting

`DHCP` No verbosity enabled
`DHCPv` Limited verbosity enabled
`DHCPV` Extreme verbosity enabled
`BOOTP` Runs BootP

IMPORTANT: After configuring the network settings and initiating the reset command, the target board will re-boot MicroMonitor. Notice that in the serial terminal interface that the new IP address is displayed; if the new IP address does not show up on the display, go back to the previous step and make the necessary corrections to `monrc`.

IMPORTANT: When using DHCP/BootP, the new IP address should be annotated in your notes, because you will need this IP address for loading GX-Linux files via TFTP. If your DHCP/BootP server assigns a different IP address every time you reboot MicroMonitor, you may want to consider manually setting the network IP address while in the development phase of your project, because the IP address is a key parameter to your TFTP communications between the host development platform and the target board.

Step 3. Modify Environment Variables

Open Visual X-Tools or another editor and edit the master `Makefile` in the *build* directory. The build directory is `gxlinux` directory created at installation time. Edit configuration variables as follows:

```
ARCH  = <arch-of-target>  (i.e., arm, mips, ppc, etc.)
BOARD = <boardxxx>        (if not set, enter the uSBC name – e.g., csb637, csb625, mx21ads, etc.)
TGTIP = <IP-address>      (enter the IP address assigned to the target board in the last step)
```

Optionally, you can modify the other configuration variables that apply, but the three most important are the ones above.

Step 4. Make All

Start an xterm/Bash shell, change directory to the *build* directory and issue the following command:

```
$ make all    (the system configures and builds everything for the target board)
```

If there are any build or install errors, make sure the toolsuite and BSP were installed correctly per instructions in Section 2. If you still have problems, contact Microcross support at +1.478.953.1907 or support@microcross.com

Optionally, you can try and troubleshoot your configuration by using the Makefile to build in incremental steps.

Step 5. Load Binaries and Startup Script

This step assumes that the target board is powered up, connected to Ethernet, and uMON prompt is displayed in a serial terminal. From the build directory in xterm/Bash, issue a make command that will load the GX-Linux binaries and startup script (we assume you assigned the target board an IP address and edited the makefile, connections are made and board is powered on).

```
$ make load
```

Three images should start loading onto the board – see the serial terminal interface for upload responses in MicroMonitor. The images loaded are as follows: 1) `zImage` (kernel); 2) `romfs.img` (ROM root filesystem); and 3) `startlinux` (startup script).

Step 6. Reset and Boot Linux

Press the board reset switch once the files have completely transferred. From the serial terminal window, issue the following command:

```
uMON> startlinux
```
(this script calls the Linux kernel and filesystem to execute)

The Linux kernel should uncompress and mount a ROM filesystem. If there are any errors, reset the board and repeat this step. Occasionally the kernel aborts on a bootup due to a corrupted flash file system. Follow steps in the MicroMonitor tips section to issue a command (uMON> tfs clean) to defrag the flash file system.

The Linux startup script is set with a Tiny Filesystem attribute that makes `startlinux` an executable like a batch file in DOS. To change this attribute so that you can automatically boot with query into Linux with a query to stop after MicroMonitor boots up, issue the following command after boot up of MicroMonitor:

```
uMON> tfs –feB cp startlinux startlinux
```

WARNING: `-feB` is case sensitive and if you use a lower case `b` in the switch option, you will not be able to boot into uMON again without re-flashing through the JTAG, which is complicated and time consuming. To change the attribute back to manual boot, substitute `-fe` in place of `-feB` in the above command above.

Step 7. Run Example Applications

When the GX-Linux boots to a Linux command shell prompt, you can enter the following commands in the serial terminal to get acquainted with all of the example programs loaded in the `/usr/bin`; each example file has a `.x` file extension. In the serial terminal command line interface window, issue the following commands:

```
# cd /usr/bin
# ls
# hello.x
```

- `# ls` (shows all examples; to execute examples, perform the next step)
- `# hello.x` (executes hello world program)

There are approximately 20 other programs that can be executed in the same manner, and each filename ends in `.x` extension. Tryout as many of the examples as you wish. Read the next section to become more familiarized with MicroMonitor.

Step 8. How to Add Your Own Applications

The fastest way to get your own application on the target board is to copy your application – named with a `.x` file extension -- into the `rootfs/usr/bin` directory, which is under the build directory, and issue the following command in the xterm/Bash shell:

```
$ make romfs   (rebuild the ROM filesystem)
$ make load    (reload your images to the target board)
```

After uploading three files to the target board (startlinux, zImage, and romfs.img), you can manually start GX-Linux by issuing the following commands in the serial terminal (uCon / Hyperterminal / Minicom):

```
uMON>                   (get a command prompt after upload)
uMON> reset             (get the board in a refresh state)
uMON> startlinux        (Linux boots to a user prompt #)
```

Alternatively, you could startlinux with NFS, so you have read/write privileges – see Section 3.2.2.

On the target board in the `/usr/bin` directory are 24 example programs that can be run from the serial console interface. Issue the following commands:

```
#  cd /usr/bin
#  ls
```

You can see all 24 examples; they have a `.x` file extension on them. To run each example, issue the by-name command:

```
#  pascal.x
#  double.x
#  float.x
#  hello.x
etc…
```

Step 9. Useful Linux Commands

```
# cat /proc/cpuinfo
```

Shows information such as bogomips, architecture, and manufacture information.

```
# cat /proc/partions
```

Shows all of the mounted device partitions with major and minor number assignments.

```
# reboot
```

Reboots system into uMON.

3.2 MicroMonitor Boot ROM Startlinux Familiarity and Tips

MicroMonitor is the boot ROM that comes with each Cogent development board. It is very useful in initializing hardware, starting serial and Ethernet services, and creating a tiny filesystem to store the embedded Linux startup binaries. Moreover, MicroMonitor facilitates loading the GX-Linux binaries using TFTP services or XModem through the serial port. We will focus on the TFTP (across Ethernet) transfer solution due to the size of the images and time it takes to load binaries.

3.2.1 Manual TFTP Up-Load Procedure

Microcross ships each board pre-configured with the Linux script, startlinux, and two binaries: zImage (kernel) and romfs.img (root filesystem and Busybox utilities). To up-load your newly created versions, follow the steps in this section. The up-load address, <target-ftp-address>, is that of the target board.

Reload startlinux Startup Script

To reload startlinux script on the target board, you have three options to control your linux bootup:

1. Load startlinux for manual bootup (requires serial connection to use the uMON command line interface to execute startlinux):

   ```
   $ cd <build-dir>/boot
   $ ttftp <target-ftp-address> put startlinux startlinux,e
   ```

2. Load startlinux for autoboot with abort query:

   ```
   $ cd <build-dir>/boot
   $ ttftp <ftp-address> put startlinux startlinux,eB
   ```

3. Load startlinux for autoboot without abort query – **warning** – there is no way to recover an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

   ```
   $ cd <build-dir>/boot
   $ ttftp <target-ftp-address> put startlinux startlinux,eb
   ```

Change startlinux Attribute Options

There is another option to change startlinux script flags without having to reload startlinux again by using the serial uMON command line interface. Microcross', startlinux, is preloaded with the e flag attribute set, which is the manual boot up setting. Two other boot options are available and can be set issuing one of the following commands:

1. Autoboot with abort query (case sensitive).

   ```
   uMON> tfs –feB cp startlinux startlinux
   ```

2. Autoboot without abort query (case sensitive) – **WARNING** – there is no way to recover from an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

   ```
   uMON> tfs –feb cp startlinux startlinux
   ```

MicroMonitor training material is available by Microcross. Here is a link to more information / documentation: http://www.microcross.com/html/micromonitor.html. Also, there is a MicroMonitor User Manual in CD-ROM `docs` directory.

Reload zImage Procedure

Similarly to the upload procedures above, you can add the `zImage` – Linux kernel – to the TFS on uMON.

```
$ cd <directory-linux-zImage-binary>
$ ttftp <target-ftp-address> put zImage zImage
```

Reload romfs.img Procedure

Similarly to the upload procedures above, you can add the `romfs.img` – root filesystem and Busybox utilities – to the TFS on uMON.

```
$ cd <build-directory>
$ ttftp <ftp-address> put romfs.img romfs.img
```

To rebuild any component of this BSP, refer to Section 5 for details.

3.2.2 Useful Startlinux Commands

A. How to manually use `startlinux` at the uMON prompt:

1) romfs.img - ROM Filesystem startup is the default

```
uMON> startlinux
```

2) nfs Root Filesystem

```
uMON> startlinux nfs <server IP address> <NFS Path>
Example: startlinux nfs 192.168.0.215 /home/gxlinux/rootfs
```

3) usb Root Filesystem

```
uMON> startlinux usb
```

4) mmc (or SD) Root Filesystem

```
uMON> startlinux mmc
```

5) cf Root Filesystem

```
uMON> startlinux cf
```

B. How to automate the use of `startlinux` scripts to autoboot the target board:

To make a startlinux custom script to autoboot, add the `eB` attribute to the file. To add this attribute from a serial terminal connection to uMON, do the following:

```
uMON> tfs -feB cp startlinux startlinux
```

A user can create a script that calls another script, too; for example, create a script named `startlinuxnfs` and put one line of code in it:

```
startlinux nfs 192.168.0.215 /home/gxlinux/rootfs
```

Copy this script into the TFS (umon Tiny File System) and set attribute to eB as described above (READ WARNING below). A host command to accomplish this follows:

```
# ttftp <TGTIP> put startlinuxnfs startlinuxnfs,eB
```

This command assumes you copy the script from the host to the TGT. A similar script can be created for USB, too, but cannot coexist with another autoboot of GX-Linux in the TFS. See user manual for details. To stop boot of Linux, press any key immediately after power on.

WARNING: `eB` is case sensitive, and if you use a lower case `eb`, you will not be able to boot into uMON again. Loading uMON will require re-flashing through the JTAG, which is complicated and time consuming -- save your backdoor boot option by NOT using `eb`.

3.2.3 Other Useful MicroMonitor Commands

`uMON> tfs ls`	List files in the Tiny File System
`uMON> tfs clean`	Defrag the Tiny File System without deleting files
`uMON> tfs init`	Delete all files and defrags the Tiny File System – requires network setup
`uMON> tfs rm <file-name>`	Delete a specific file named <file-name>
`uMON> help`	Shows all of the available commands
`uMON> help <command-name>`	Shows specific help on command name
`uMON> set`	Shows all settings in memory
`uMON> flash info`	Shows all flash sectors and which ones are locked and erased
`uMON> flash unlock x-y`	Unlocks sectors x (lower bound) to y (upper bound)
`uMON> flash erase x-y`	Erase sectors x (lower bound) to y (upper bound)

The MicroMonitor User Manual is located in the docs directory on the GX-Linux CD-ROM and also on the Microcross website. The Microcross website URL for updates to MicroMonitor source code and user manual are located here: `http://www.microcross.com/html/micromonitor.html`

3.3 Sound Example with SOX (if supported by the target board)

Issue these commands after GX-Linux boots and the serial console is live:

```
# cd /opt/snd-example
# cmix vol on max
# sox  -b  -u  -r  8000 -c  1  microcross.wav  -t  ossdsp  /dev/dsp
```

Listen to the audio file.

You can use another audio mixer called aumix:

```
# aumix  -v  +100
# sox  -b  -u  -r  8000 -c  1  microcross.wav  -t  ossdsp  /dev/dsp
```

All programs (sox, aumix, and cmix) can display their help screen with the command and '--help' to display options. e.g.

```
# sox –help
```

3.4 Telnet Usage with Target Board

The BusyBox utilities are setup for a Telnet configuration. In `/etc/init.d/rcS` is one line that instructs the system to run `telnetd`; the line is `telnetd –l /bin/sh` and can be disabled by removing or commenting out the line in `rcS`.

To invoke a Telnet session, simply open a shell on your Linux host computer and input the following:

```
# telnet <board-ip-addr>
```

Section 4.0 Visual X-Tools Example Application Project

The following steps describe the operation of the Visual X-Tools with an example Embedded Linux program for the target board. The example shown can be replicated for any of the other examples described below. The example program was created with the Visual X-Tools wizard.

4.1 Hello Example Program

This section briefly goes over the standard hello world program that talks to the console via libc-xflat.so using printf statements.

Step 1.

Startup Visual X-Tools IDE from a desktop icon or Bash/xterm shell. See the Visual X-Tools IDE icon in Figure 4.1.

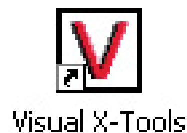


Figure 4.1 Visual X-Tools IDE Icon

Step 2.

Click on `Project|Open Workspace…` from the menu. Navigate to the following directory:

`../<build-directory>/examples/vxtools-hello`

Open the workspace file `hello.vxw` and go onto the next step.

Step 3.

On the Visual X-Tools menu click on `Build|Set Active Configuration…` and select either `Debug` or `Release`.

Step 4.

On the Mini Toolbar (see Figure 4.2), click on `Clean & Build Project` icon.

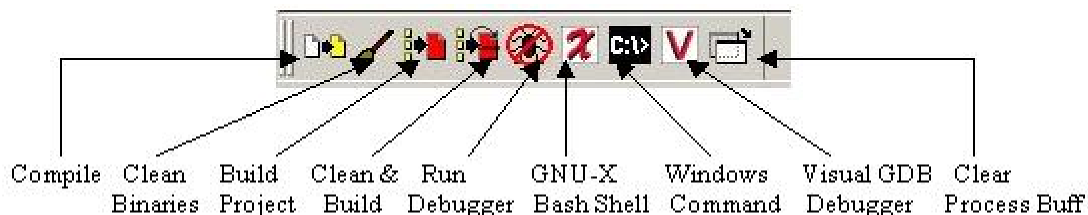


Figure 4.2 Mini-Toolbar Description

These commands are also located under the `Build` menu. The Build window at the bottom of Visual X-Tools shows the verbose output of the build. After completion of the build, go to the next step.

Step 5.

If you selected the `Debug` Active Configuration in Step 3, then click on `Build|Copy Debug Image to Examples Dir.`

If you selected the `Release` option for Active Configuration in Step 3, then click on `Build|Copy and Strip Release to a Binary Image`, then click on `Build|Copy Release Image to Examples Dir`.

Step 6.

Issue the following commands in xterm/Bash shell for rebuilding the `romfs.img` filesystem and replacing the existing version on root filesystem.

```
$ make install-apps romfs
$ make load
```

To get more specific information on how to use Visual X-Tools, consult the Visual X-Tools User Guide.

--- END OF EXAMPLE PROGRAM ---

Section 5.0 Making Modifications to GX-Linux

Every developer will have to make at least one or more modifications to GX-Linux, and depending on which Edition you are working with, the steps may vary slightly.

5.1 Modifying GX-Linux Enterprise Edition

5.1.1 GX-Linux BSP Components

The software downloaded to the target hardware is comprised of various components, listed below:

- Bootloader (uMON)
- Linux kernel (Linux)
- Target file system

The target file system itself consists of various components:

- Linux file system framework including glibc libraries
- Development applications (Busybox, tinylogin, tftp, etc)
- Loadable kernel modules
- Other applications

5.1.2 Building All of the Components

All of the GX-Linux BSP components can be built using the following commands:

```
$ cd <build-directory>
$ make all
```

glibc (Ent): zImage (~1.2Mb), romfs.img (~4.3Mb), and startlinux (~1.3 kb) – size will vary as apps apply.

5.1.3 Building All Components Excluding Examples

All of the examples can be built using the following commands:

```
$ cd <build-directory>
$ make distclean
$ make config system
```

The above step builds everything. To add your own example applications after building the system, follow the procedure below, *Building Your Own Applications*.

5.1.4 Building Your Own Applications

We assume you already performed at least a preliminary build of the kernel, library and root filesystem with BusyBox. You can build your own applications, load them into `/usr/bin` (or any other directory) and run them on the target uSBC:

- Build your application(s) and name the binary executable with a `.x` file extension (recommended).
- Copy your application files into the `<build-directory>/rootfs/usr/bin` directory.
- Re-build the ROM filesystem, and reload the target board via TFTP (assuming the target board is powered on and connected to your network), issue the following commands:

```
$ cd <build-directory>
$ make romfs
$ make load
```

If you have any load errors, make sure your target IP address is correct in the Makefile (located in build directory).

5.2 Changing the Kernel Configuration

5.2.1 Configuration Files

The Linux kernel supplied with your BSP is pre-configured and ready for use. If your project requires components that are not contained in the default GX-Linux kernel, you may want to change the kernel configuration. The kernel configuration is expressed in two files: (1) the `.config` *working copy* and (2) `kernel-conf` *master backup copy*. The working copy is retained at `.../gxlinux/linux/.config`; the master backup copy at: `.../gxlinux/config/<csbxxx>/kernel-conf`.

The configuration file is read and modified by xconfig, described below. After running xconfig, another derived file, `.../gxlinux/linux/include/linux/autoconf.h` is created. The autoconf.h file is included by most kernel source files to control which features are built.

5.2.2 Changing the Working Configuration

Changing the GX-Linux kernel configuration requires changing the .config working copy. It should not be modified directly, but rather it should modify it using the steps below. In this procedure, we use `make ARCH=<arch> xconfig` to modify .config. Note: We assume that you have the QT components from the KDE development package installed before using xconfig.

1. Type the commands below to initiate xconfig:

```
$ cd <build-directory>/linux
$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

   Or alternatively:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- xconfig
```

   WARNING: By leaving the ARCH off of “make menuconfig” or “make xconfig”, the system will default to x86 configuration and saving this configuration will corrupt your kernel build environment (goto 5.3.2 if this happens to restore your pristine kernel configuration).

2. The menuconfig/xconfig will display all of the kernel options and their current settings. Menuconfig/xconfig will allow you to change any kernel setting. Be sure to save your changes before you exit from menuconfig or xconfig. With the menuconfig/xconfig session complete, you can build the new GX-Linux kernel as follows (assuming arm for purposes of this example):

```
$ cd <build-directory>/linux
$ make ARCH=arm CROSS_COMPILE=arm-linux- oldconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux- vmlinux
```

   If the build is successful, the resulting kernel image can be found in the Linux directory:

   `<buld-directory>/linux/vmlinux` -- an ELF object format

3. After confirming a proper build of vmlinux, perform the simple update of the master config file.

```
$ cd <build-directory>
```

```
$ make SaveCfg
```

   (saves the `.config` file to the master config file to the `config/<xxxxxx>/kernel-conf` file)

Now you can use the master configuration and build system with confidence that the kernel will build. Go to the root build directory and perform the usual 'make config' and 'make system' to get everything built as usual.

5.2.3 Resetting the Kernel to the Default Configuration (disaster recovery)

Should you want to return to the GX-Linux default settings, you can accomplish this with the steps below (assuming the ARM in this example, but can be extended to MIPS and others):

```
1. $ cd <build-directory>
2. $ make distclean
3. $ cd linux
4. $ make ARCH=arm CROSS_COMPILE=arm-linux- mrproper
5. $ cp arch/arm/configs/<name_of_defconfig> .config
6. $ make ARCH=arm CROSS_COMPILE=arm-linux- oldconfig
7. $ make ARCH=arm CROSS_COMPILE=arm-linux- vmlinux
8. $ cd ..
9. $ make SaveCfg

Note: In step 5 above, you need to insert your architecture if different than
arm and the name of the _defconfig you are using. Some examples are as follows:
         csb637_defconfig
         csb535fs_defconfig
         csb735_defconfig
         mx21ads_defconfig
         mx21adse_defconfig
```

5.2.4 Customizing the Kernel Configuration

If you find that your particular needs cannot be met using the available kernel configuration options, you may want to modify the make xconfig scripts that control the configuration session. The particular file that controls the xconfig program presented above is at:

```
…/gxlinux/linux/arch/arm/config.in
```

5.2.5 Modifying Kernel Loader Address

If you want to specify exactly where you want your kernel to load and make everything behave relative to this load address, there is one parameter that has to be changed in the kernel source tree. Here is a summary of the changes for four GX-Linux target architectures currently in use:

ARM / XScale	**Location in Source Tree**
Machine Type AT91RM9200	`gxlinux/linux/arch/arm/mach-at91rm9200/makefile.boot`
Machine Type i.MX	`gxlinux/linux/arch/arm/mach-imx/makefile.boot`
Machine Type PXA	`gxlinux/linux/arch/arm/pxa/makefile.boot`

5.3 Changing the Target File System

The directory structure under …`/gxlinux/rootfs` is the master file system structure used to create the target file system. Files can be installed into the target file system by simply copying them into the appropriate sub-directories under …`/gxlinux/rootfs`. The target file system is built by default as a compressed `romfs.img` file system using genromfs.

5.3.1 Adding Content to the Target File System

You can configure the target file system with additional content not included in the BSP's default target file system. The following steps illustrate two methods that can be used to put new applications into the target file system. In these illustrations, the application is the "Hello, World" example.

1. The following steps compile and install the "Hello, World" example into the target file system image on the Linux development workstation.

```
$ cd <build-directory>/examples/hello
$ make apps install-apps romfs
```

   The make `romfs.img` target command creates the target file system image as `<build-directory>/romfs.img`, which is the file we downloaded to the target hardware using the uMON bootloader.

5.4 Putting the New System on the Target Hardware

5.4.1 Putting the New System on the Target Hardware

Once you have created a new BSP component (a new kernel image, a new file system image, and the uMON `startlinux` script), the uMON bootloader can be used (a) to download the components into the target hardware's SDRAM, (b) to save the component in FLASH memory, and/or (c) to boot the new system. The steps below describe the general procedure to accomplish this. See the uMON documentation or Section 3 of this User Guide for more detailed instructions.

1. Power cycle the target hardware. Start your serial terminal that is connected to the target board, and if configured correctly, you should get the uMON prompt.

2. From xterm/Bash shell, change to the *build* directory and issue `make load` to download the new kernel, file system, and startlinux script to the TFS.

3. Issue the `startlinux` script on the uMON command line in the serial terminal.

   The `startlinux` script is a uMON script that executes the required uMON commands to boot the Linux kernel with the right configuration. The configuration values in `startlinux` were *auto-generated* by the GX-Linux BSP Configuration Tools according to the values selected using the BSP configuration tool.

   The Linux kernel and file system image must have been saved to TFS prior to executing the `startlinux` script. Further, the `startlinux` script will require that the Linux kernel and file system image have the same names in TFS as they did on the Linux workstation. The `startlinux` script should be stored on TFS with the "executable script" attribute enabled (the `make load` command automatically handles this).

5.5 Read-Only and Read-Write File Systems

5.5.1 ROMFS File System

The default file system used in the GX-Linux BSP is not a permanently writable file system. Anything written to the romfs will be lost when the target hardware is powered down. Compact Flash, SD/MMC, USB Pendrives and other permanent memory storage devices can be used as permanent storage for your filesystem and/or data. You can also create an NFS mounted root filesystem on your network and simply access it using the `startlinux` script or at Linux run-time.

5.5.2 NFS Mounted Root Filesystem

To mount an NFS read-write root filesystem, you need to have a server setup for NFS mounting. The procedures to perform this task are beyond the scope of this User Guide and should be done by a competent Linux system administrator. The administrator needs to create the NFS server in a location on your network that you can place your `rootfs` directory on and have the correct dev directory. When you `make all`, two sets of devs get created: 1) initial set of `pseudodevs` for `romfs.img`; and 2) final set of `devs` for `rootfs` to become a NFS mountable root filesystem. The pseudodevs are all prefixed with `@` symbol, while the real devs are not. Two pieces of information are important to use GX-Linux with an NFS mounted filesystem: 1) the NFS server IP address; and 2) the NFS `roofs` path on the server. Recall that `rootfs` is part of the build directory in `gxlinux`. You can copy this directory after `make all` and put it on another machine for mounting to, or you can simply make your host system an NFS server and add the path to `rootfs` in the GX-Linux build directory.

Once the target filesystem is mounted via NFS, then applications can be written or modified under this filesystem, which is visible after the next power cycle. In this scenario, the target filesystem is not hosted in the target hardware but rather on a hard drive of a remote Ethernet connected machine, typically your Linux development workstation.

How to Mount Using an NFS Server after Bootup Using the romfs.img

Issue the following command:

```
# mount –t nfs –o nolock <ip-address>:/<dir-to-mount> /mnt/nfs
```

Example

```
# mount –t nfs –o nolock 192.168.0.2:/home/james/gxlinux/rootfs
                                                          /mnt/nfs
# cd /mnt/nfs
# ls
```

How to Start GX-Linux with NFS Mounted Automatically or Manually from the Startlinux Script

Setup of Server
There are many versions of Linux and GUI environments to interface with. The example we show uses Red Hat Enterprise Workstation v3/4. Here are the steps to setup an NFS server on the host machine (assuming a Linux OS is booted and running Red Hat Enterprise v3/4).

Step 1. Click on 'Main Menu|System Settings|Server Settings|NFS'
Step 2. Enter root password
Step 3. Click on 'Add'
Step 4. Enter the 'Directory' path or browse to it: e.g., /home/james/gxlinux-535/gxlinux/rootfs
Step 5. Enter host(s) address with netmask as follows: e.g., 192.168.0.0/255.255.255.0
Step 6. Click on basic permissions: 'Read/Write'
Step 7. Click 'OK'
Step 8. Click on menu 'File|Quit'
Step 9. Click on 'Main Menu|System Settings|Server Settings|Services'
Step 10. Enter root password
Step 11. Click on 'NFS" and 'Restart', which are on the same dialog page; NFS should report started successfully
Step 12. Click on 'File|Quit' and resume your startlinux activity
e.g., umon> startlinux nfs 192.168.0.113 /home/james/gxlinux-535/gxlinux/rootfs

The developer has a number of options as to how to automate the mount of an NFS server, but the most common way with GX-Linux is to setup environment variables in uMON to point to the server IP and path, and then issue the `startlinux` command or build an automated script and set it with autoboot with query attributes.

1. Add two monrc Environment Variables: NFSSERVER and NFSPATH
   - `uMON> set NFSSERVER <IP-ADDRESS>`
   - `uMON> set NFSPATH /path/to/rootfs`
   - `uMON> set –f monrc`
   - `uMON> reset`
2. Manually Start Linux from Serial Terminal
   - `uMON> startlinux nfs $NFSSERVER $NFSPATH`
3. Autostart Linux from uMON Bootup
   - `uMON> edit –feB startlinuxnfs`
   - `uMON> i`
   - `uMON> startlinux nfs $NFSSERVER $NFSPATH`
   - `uMON> .`
   - `uMON> q`
   - `uMON> reset` (autostarts with query into linux with NFS mount)
   - `# reboot` (reboot from Linux to query uMON – hit any key to stop Linux boot)
4. Modify Autostart File Attributes to Manual Start Attribute with TFS Copy Command
   - `uMON> tfs –fe cp startlinuxnfs startlinuxnfs` (the –fe gives the file an attribute for executing a script)
   - `uMON> reset`
   - `uMON> startlinuxnfs` (start Linux nfs mode manually)

The procedure in 3 and 4 above can be done also for a `startlinux` without NFS; simply omit `nfs $NFSSERVER $NFSPATH` from the command line in the Autostart procedure. If there are any errors, consult your system administrator for help, or manually modify `startlinux` in the …`\config\$(BOARD)\` directory on the host workstation to hard-code your NFS settings (NFSSERVER and NFSPATH); reload the script into the uMON TFS (tiny file system), and then issue the command in step 2 above.

5.5.3 Writing to On-Board Flash via a Pendrive (USB) – If Supported on Target

Some Linux applications will require the ability to store data persistently in flash. Such applications will require extensions to the target hardware to allow hosting of a read/write flash file system. These extensions will include:

- Provision of a writable flash file system that the kernel can mount on bootup. Writable flash file systems are available under the 2.6 kernel (ext2, ext3, JFFS, JFFS2, VFAT, etc.). These flash file systems are well documented and discussed in the Linux Open Source embedded systems community.

- Kernel configuration changes (for example, via make xconfig or menuconfig) to enable the particular file system type used.

- Addition of an underlying flash media access layer/driver to support the board's flash chip(s) or external compact flash pendrive or USB drive.

To mount a pendrive using the standard configuration of GX-Linux and USB host. Insert the pen drive and the operating system will echo the mount point; issue the following command:

```
      # mount  -t vfat /dev/sda1  /mnt/pendrive
or alternatively:
      # mknod /var/sda1 b 8 1
      # mount –t vfat /var/sda1 /mnt/pendrive

for a standard ext filesystem, issue the following command:
      #mount /dev/sda1/  /mnt/pendrive
or alternatively:
      # mknod /var/sda1 b 8 1
      # mount /var/sda1 /mnt/pendrive
```

After mounting the pen drive, you can change to this drive or copy files to it using the mount point `/mnt/pendrive`. Use the `umount /mnt/pendrive` command to un-mount the drive.

5.5.4 How to Prepare a Pendrive with a Root Filesystem and Work with GX-Linux

Start an xterm or Bash shell and log in as root or `su`. We assume that `sda1` is the USB device in this example. Insert the pendrive into your host computer that has GX-Linux installed. Mount the pendrive.

Example

`# mount /dev/sda1 /mnt/pendrive` (create the pendrive directory in `/mnt` if it doesn't exist)
`# df` (reports back the number of blocks for all physical devices)
Note the number of blocks reported for `sda1` or your USB device name.
`# umount /mnt/pendrive`
`# /sbin/mkfs /dev/sda1 <#blocks>`
Ask your system administrator how to format filesystems if you are not sure how to use this command.
`# mount /dev/sda1 /mnt/pendrive`
`# cd /mnt/pendrive`
`# cp –R /<path-to-rootfs>/gxlinux/rootfs/*  .`
`# ls dev` (if this directory is empty, then you do not have any device nodes, read below)

Make sure `rootfs/dev` directory is populated with device nodes, because they get deleted when a `distclean` is run. If `dev` directory is empty, then issue a `make all` in GX-Linux before issuing the above command.

```
# umount /mnt/pendrive
```

Now pull the pendrive out the host machine and with power off the target board, insert the USB pendrive into the USB host socket and power on the target board (assuming serial and Ethernet are connected). At the MicroMonitor umon prompt, issue the `startlinux usb` command (assuming you have GX-Linux files already loaded). The pendrive should be mounted as your root filesystem; if not, then troubleshoot the pendrive on your host computer to see if the root filesystem looks like the same one in your rootfs directory. Follow the procedures in previous sections to get NFS working to validate the root filesystem.

5.5.5 Writing to On-Board Flash via MMC/SD Card – If Supported on Target

Check to see if `mmc` is available in the device driver `devices` list:

```
# cat /proc/devices
```

If `mmc` is listed on the Block devices, then you may issue the following command sequence manually or add to the `/etc/init.d/rcS` file in rootfs and varetc to be executed automatically at startup. The `rcS` file is like an `autoexec.bat` file in DOS that executes commands as given at startup. The `microcross-banner` is executed in this manner.

Th `mmc` block device will have a number associated with it when the above procedure is executed, and the major number is used in making a node. WARNING: Connect CF/MMC device before powering board on.

Example:

```
# mknod /var/mmcblk0p1 b 254 1
```
or alternatively
```
# mount –t vfat /var/mmcblk0p0 /mnt/mmc
```
or alternatively
```
# mount –t vfat /var/mmcblk0p1 /mnt/mmc
```

Leave the `-t` off for ext2 and ext3 filesystems.

If the `/dev` directory in rootfs has a node for MMC (i.e., `mmcblk0p1` or `mmcblk0p0`), then the mount can be done in one step.

`# mount /dev/mmclblk0p1 /mnt/mmc` (as an example)

For some Cogent boards, the `startlinux` script has the feature to mount a root filesystem that has been created on an SD or MMC card. To automount this filesytem, issue a `startlinux mmc` command in uMON. Optionally, you can create another startlinux script, say `startlinuxmmc`, and put on line of code in it to call `startlinux` with the `mmc` argument. When TFTPing the new script, add the`eB` attribute to autostart Linux with query from uMON.

5.5.6 Writing to On-Board Flash via a Compact Flash Card – If Supported on Target

The Compact Flash (CF) socket on Cogent boards is configured as a mass storage device (IDE) by default with GX-Linux and are not hot swappable. WARNING: Connect CF card before powering board on. Contact Microcross for information on other options available (e.g., PCMCIA, WIFI, Wirelss Ethernet, etc.).

The compact flash card driver can be mounted with the following command:

```
# mount /dev/hda1 /mnt/cf
```

Note: We have seen in our testing that some CF cards 512 MB and larger do not work in the Cogent boards. Recommend using 256 MB or smaller memory cards or trying out various cards before committing to use one in your design.

Section 6.0 Debugging Applications

There are two common tools used for symbolic debugging software used in Linux based target hardware. The GDB debugger is typically used to debug applications. A JTAG emulator is typically used to debug the Linux kernel and kernel device drivers. Debugging using a JTAG emulator is dependent on the emulator used, and thus is not discussed in this manual.

6.1 The GDB Debugger

6.1.1 GDB Debugger

The purpose of a debugger is to allow the developer to see what is going on inside his/her program while it executes. Two debuggers are provided in the GX-Linux BSP, and they are located in `/usr/bin`; the toolnames are `<target-alias>-gdb` and `<target-alias-gdbtk` (e.g., arm-linux-gdb and arm-linux-gdbtk). GDB is the command line debugger, and GDBTK is the Visual GDB (aka Insight). Use of that debugger is the topic of this section. For detailed information regarding usage of the GDB debugger, please refer to one of the following:

- The info and standard GDB man pages included on your Linux workstation.
- The online manual provided by the Free Software Foundation at http://www.gnu.org/software/gdb/documentation.
- Debugger docs on the GX-Linux CD-ROM under docs directory

6.1.2 gdbserver

The GDB debugger executes on the user's Linux workstation. In order to communicate with a program running on the target hardware, an additional program must run on the target hardware. This program is gdbserver. It is included with the GX-Linux toolchain and is installed in the target file system as `/usr/bin/gdbserver`.

6.1.3 gdbserver Communications

There are two methods for communication between the target hardware's gdbserver and the Linux workstation GDB client. One method uses an available serial port while the other uses the Ethernet port.

Some target hardware have multiple serial connectors with one of them dedicated to the system console (typically `/dev/ttyS0`) and another (`/dev/ttyS1`) is available for GDB communication with the Linux workstation. In the case where the target hardware has only one serial port, it is probably not available for GDB debugging. In this case, the Ethernet port should be used for GDB communication with the Linux workstation.

If you are going to use the serial port for debugging, see the 6.4 Changing Serial Port Settings section for additional information about 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.

6.1.4 Graphical Front-Ends

If you would like to use a graphical front end with arm-linux-gdb, there are several compatible front ends available. One popular front end is the ddd utility (http://www.gnu.org/software/ddd). Below is an example of how you would invoke the ARM9 arm-linux-gdb debugger, using the ddd graphical front end:

```
$ ddd -debugger arm-linux-gdb
```

6.1.5 Debug-able Files

You can use the `<target-alias>-gdb` debugger (with the gdbserver) to debug Linux programs on the target hardware. You can use any program invoked from the Linux command line with this debugger, *provided that* the program was built to be compatible with GDB.

6.1.6 Visual GDB

To debug using the Microcross Visual GDB (a.k.a. Insight), refer to the Microcross documentation, which is offered online at www.microcross.com.

6.2 Debugging Examples

To quickly get started debugging, several example files were included in the GX-Linux package. Note that gdbserver has an issue in debugging `start` files, and we have confirmed this issue with four versions of gdbserver. An error code, `illegal instruction`, is the symptom. To work around, set a breakpoint past `start` functions, and single step from there. A list of all the examples follows:

6.2.1 Example with Visual GDB

Step 1. Start gdbserver on the Cogent uSBC

Using the Minicom interface to Linux running on the Cogent uSBC, issue the following command:

```
# gdbserver localhost:2001 /usr/bin/div.x
```

Step 2. Start Visual GDB

Issue the following set of commands in xterm shell (the directory information is notional – use your path setting).

```
$ cd /home/steve/gxlinux/examples
$ arm-linux-gdbtk div.x
```

Click on `File|Target Settings…` and select `GDBserver/TCP` from the drop-down list. Enter the `Hostname` – the Microcross pre-assigned host is the IP address: `192.168.0.110`. Next, enter the `Port` number, and the Microcross port number assigned is `2001`. Click `OK` and click on the `Run|Connect target` menu item. Set your breakpoints and click on the `Run` icon. You should now be able to single step through the example program.

6.2.2 Example with GDB

Debugging the program: Once you have placed your executable "ELF" program in the target hardware's file system, it is ready for debugging using the following steps. On the Linux workstation, you will need the same, debug-able "ELF" version of the program in the same directory where you will invoke arm-linux-gdb.

1. **Start the gdbserver.** On the target hardware, invoke the gdbserver in the directory where you have the file you are debugging: the "flat" or "xflat" format version of your program. To start the target hardware-side of the GDB session, do one of the following:

   If you are using a network GDB connection, enter the command below:

   ```
   # cd /usr/bin
   ```

```
# gdbserver localhost:5472 hello.x
```

The port number 5472 was selected randomly as a high port number. Any high port number that doesn't conflict with existing active ports will work. The command line also indicates the name of the program you intend to debug, hello in this example. This program is the debug-able "ELF" program that we built and installed above.

If using a serial GDB connection, enter the commands below. See the Changing Serial Port Settings section for instructions on building the stty command into busybox.

```
$ stty -F /dev/ttyS1 -iexten -echoctl 38400
$ <target-alias>-gdb /dev/ttyS1 hello.x
```

Example: In either case, gdbserver will respond with something like the following example:

```
# cd /usr/bin
# /usr/bin/gdbserver localhost:5472 hello.x
Process hello created; pid = 26
```

2. **Connect <target-alias>-gdb.** Start the Linux workstation-side of the GDB session by invoking the <target-alias>-gdb debugger. This establishes a connection to the GDB server previously started on the target hardware. We invoke <target-alias>-gdb while in the directory containing the debug-able version of our "ELF" program file, hello. Then issue commands at the GDB command line prompt to establish the connection and load the program's debug symbols.

   If you are using the `ddd` graphical front end, invoke it using the following command.

   ```
   $ ddd -debugger <target-alias>-gdb
   ```

   If using a network GDB connection, enter the commands below.

   ```
   $ cd <build-directory>/examples/hello.x
   $ <target-alias>-gdb
   (gdb) set endian little
   (gdb) target remote x.x.x.x:2001
   (gdb) add-symbol-file hello.x
   ```

   Where:

   - The string *x.x.x.x* is replaced with the actual IP address of the target hardware. (If you are not sure what the IP is, type `ifconfig` on the target hardware command line to generate a network configuration report).
   - The port 2001 was selected randomly as a high port number. It must match the port number that was provided on the preceding target hardware gdbserver command.

   If using a serial connection to the gdbserver, enter the commands below.

   ```
   $ cd <build-directory>/examples/hello.x
   $ <target-alias>-gdb
   (gdb) set endian little
   (gdb) set remotebaud 38400
   (gdb) target remote /dev/ttyS1
   (gdb) add-symbol-file hello.x
   ```

   By default, most Linux workstation's serial ports start up at 9600 baud. On the target hardware `/dev/ttyS1` starts up at 9600 baud. See the Changing Serial Port Settings section for instructions

on building the stty command into busybox which will allow you to change the baud rate on `/dev/ttyS1`.

6.3 Debugging Using the .gdbinit

Using ~/.gdbinit. The <target-alias>-gdb/gdbtk debuggers will read an optional `.gdbinit` when they are started -- must be in the same directory where starting gdb/gdbtk. For either gdbserver network or serial connection, the corresponding set of GDB commands can be defined as a macro within your `.gdbinit` to simplify these steps.

Consider the following sample `.gdbinit`:

```
define target_remote
  set endian little
  target remote x.x.x.x:2001
end
```

With this macro defined in your `.gdbinit` file, GDB can be connected to the remote gdbserver with the single command:

```
(gdb) target_remote
```

6.4 Debugging Linux Kernels and Drivers with the BDI2000

Note: Microcross is a distributor for Abatron in the US and Canada. This JTAG debugger is by far the most superior performer for debugging Linux kernels and drivers using the GNU debugger (GDB / Visual GDB), and Microcross has done a fairly extensive job in documenting the procedure below. To find more information, visit the Microcross website under the solutions/debugger menu or call sales directly at +1.478.953.1907.

BDI2000 CONFIGURATION

You have to use the MMU XLAT (Not needed for ARM or XSCALE) option in the config file or you will not be able to debug the Linux kernel once the virtual addressing is started. For the purposes of purely debugging Linux you don't need to initialize the memory controller or have any register initializations in the config file since Micromonitor bootloader will setup all the necessary registers. To let the Micromonitor bootloader perform all the initializations and use STARTUP RUN in the [TARGET] section of the config file.

In some cases you might need to write some register values before you will be able to debug or bootup your kernel. So start out by creating a copy of your config file and empty out the [INIT] list in the config file. This is important that you comment out the [INIT] settings because you may end up with conflicts with values of your bootloader initializes with. The effects of these conflicts are mostly not obvious until you start debugging. The config file should disable the watchdog timer if you have one and perform only the minimal register writes needed to enable proper booting of your kernel. Use STARTUP RESET to force the BDI2000 to process the [INIT] list if necessary.

HOST PC CONFIGURATION

On the host system, the BDI2000 should be able to read its config file from the TFTP server, and you should have already installed GNU X-Tools. GDB can be run natively on Linux or you can run on a Windows based PC with the Microcross version of Cygwin.

LINUX CONFIGURATION

You will need to make two separate Linux kernel objects. GX-Linux already does this for the Cogent ARM and XScale targets, and the file names are `zImage` (located in …`/gxlinux/linux/arch/arm/boot` directory) and

vmlinux (located in .../gxlinux/linux directory). In all cases, the target kernel image is stripped of debug symbols (and in the case of ARM/XScale, the image is compressed). The kernel image size will be between 1.5 and 3 MB. The kernel image in …/gxlinux/linux directory is used for source level debugging and will have the entire debug symbol library built in it once the kernel is configured for debug symbol generation. This debug kernel version, named vmlinux, will be much larger, 20 to 30 MB in size than the zImage. Without the vmlinux and debug symbols, source level debugging is not possible. To add symbols for debugging to the vmlinux file, you will need to run the kernel configuration utility to enable kernel hacking and Compile the kernel with debug info. Here is an overview of the steps using GX-Linux:

- Open an xterm/Bash shell on your Linux host.
- Change directory to …/gxlinux/linux.
- Issue make ARCH=arm menuconfig
- Exit and save the configuration.
- Issue the kernel make commands: make clean and then make.
- When finished, load the zImage to the target board with ttftp utility that Microcross provides (e.g., ttftp <target-ip> put zImage zImage) – of course you can only issue this command from the same directory where the image is located, and we assume the target board is configured with an Ethernet static IP address and that the target is connected and powered on.

TARGET SPECIFIC ISSUES

In general writing to/clearing debug registers via software will cause the BDI2000 to lose any breakpoints it has already set. Hence, make sure your Linux Kernel does not modify the debug registers while you are debugging with the BDI2000. Also, if you keep hitting exceptions in your kernel or keep breaking at interrupts, you will need to enable VECTOR CATCH 0xAddress in the BDI2000 config file. Moreover, you need to use STARTUP RESET for this to work properly.

XScale (CSB625 config issues)

The BDI2000 Vector Table cannot be written to by software while there is a JTAG Emulator connected to an XScale target. You have to manually update the Vector Table with the BDI2000 through the config file to values expected by the kernel, otherwise the kernel will hang at bootup.

Here is a snapshot of the [TARGET] section of the config file for the Cogent CSB625. The [INIT] section is entirely commented out to allow bootup using Micromonitor initialization.

```
[TARGET]
CPUTYPE          PXA250          ;the target CPU type
JTAGCLOCK        1               ;use 16 MHz JTAG clock
DBGHANDLER       0xFFFF0800      ;debug handler base address
VTABLO 0         0xEAFFC00E
VTABLO 1         0xEA0000E9
VTABLO 2         0xEA000108
VTABLO 3         0xEA0000C7
VTABLO 4         0xEA0000A6
VTABLO 5         0xEA000169
VTABLO 6         0xEA000128
VTABLO 7         0xEA000147
ENDIAN             LITTLE        ;memory model (LITTLE | BIG)
VTABHI 0          0xEF9F0000
VTABHI 1          0xEA0000DD
VTABHI 2          0xE59FF410
VTABHI 3          0xEA0000BB
VTABHI 4          0xEA00009A
VTABHI 5          0xEA0000FA
```

```
VTABHI 6          0xEA000078
VTABHI 7          0xEA0000F7
ENDIAN            LITTLE          ;memory model (LITTLE | BIG)
BREAKMODE  SOFT                   ;SOFT or HARD
STARTUP RUN
```

SELECTING A BREAKPOINT LOCATION

To be able to debug the Kernel after the MMU turns on, you have to set the initial breakpoint to a location where the MMU is already active. We suggest setting a breakpoint at 'start_kernel' as our symbol to break on. You should connect with GDB after you have stopped the Kernel at 'start_kernel'. If you connect to GDB before the MMU is enabled, the BDI2000 will give out address errors. The location of 'start_kernel' varies from each version of Linux kernel and each target board. To find all of your symbol addresses, look at your symbol map in `…/gxlinux/linux/System.map` file. You can see where `start_kernel` is located. Alternatively, you can issue the command: `grep start_kernel System.map`.

PROCEDURE FOR DEBUGGING LINUX KERNEL WITH BDI2000 & GDB

Assumptions

- BDI2000 is setup with a working config file and static IP address
- GNU X-Tools™ is installed on the host machine
- GX-Linux™ BSP has `zImage` and `vmlinux` (w/debug symbols) pre-built
- Target board is available with power supply, serial cable, and Ethernet cable

Steps

1) Connect the BDI2000 to the target board.
2) Start two shells and a serial terminal application:

   - Start an xterm/Bash or DOS (if on Windows) shell for creating a telnet session to the BDI2000 -- call this Shell 1.

   - Start an xterm/Bash or GNU X-Tools (if on Windows/Cygwin) for connecting GDB to the target -- call this Shell 2.

   - Start a serial terminal program (Minicom / Hyperterminal / uCon) to connect to the target board at 38,400 8N1 settings.
3) Power-up the BDI2000 first and start a telnet session in Shell 1; let the BDI2000 load the config file, and when it says waiting for target VCC, power-up the target board.
4) At this point the default behavior is reset halt, which means the target will not be running. Enter `i` at the BDI2000 prompt, to make sure the target is not running then.
5) Type `go` at the BDI2000 prompt and watch the target bootup through the serial monitor.
6) If not already loaded, then load the zImage into Micromonitor's TFS (Tiny File System) using the `ttftp <target-ip> put zImage zImage` command, but do not start booting Linux.
7) Type in `halt` at the BDI2000 prompt and set a breakpoint at `start_kernel`:

   `BDI> bi 0xCxxxxxxx` (substitute the actual address)
8) Issue a `go` command to give control back to the target: `BDI> go`
9) Issue the command `startlinux` at the Micromonitor prompt: `uMON> startlinux`

10) After a few seconds the BDI2000 prompt should read `target has entered debug mode`; enter `i` at the BDI2000 prompt and confirm that you have stopped at `start_kernel` address. Issue a BDI2000 command `ci` to clear the breakpoint: `BDI> ci`
11) Issue a few `ti` commands from the BDI2000 prompt to confirm that you can single step through the target. The BDI2000 prompt will tell you that it is stepping: `BDI> ti`
12) On the host platform in Shell 2, change directory to …`/gxlinux/linux` and start GDB. The command is `arm-linux-gdb`.
13) Issue a command to load the debug symbol table:

    ```
    (gdb) file vmlinux
    ```
14) Connect the GDB to the target using the following commands in Shell 2:

    ```
    (gdb) target remote 192.168.0.131:2001
    ```

    Where `192.168.0.131` is a notional BDI2000 IP address and `2001` is the default port number.
15) Perform a few steps using `stepi` command in the GDB prompt; you should see a step occur in the BDI2000 Prompt:

    ```
    (gdb) stepi
    (gdb) stepi
    ```

    Also perform a `list` command in the GDB prompt and make sure GDB knows where it is:

    ```
    (gdb) list
    ```

    or a disassemble command:

    ```
    (gdb) disassemble
    ```
16) Finally, set your own break points in GDB and see if it the BDI2000 hits the breakpoints:

    ```
    (gdb) b <symbol_name>
    (gdb) c    (continue to run until a breakpoint is hit or to program end)
    ```

PROCEDURE FOR MODULE DEBUGGING

Before beginning, make the module and make sure it is accessible by the root file system on your target. Also be sure to compile the module with the `-g` option so that the paths to the source code is known. Follow the steps below:

1) Bootup Linux on your Target Machine with the BDI2000 connected then
2) Do issue an `insmod` on the target:

    ```
    [target]# insmod -f -m module.o > module.map
    ```

    This creates a map file you'll use latter. Note the major number that is printed when you do `insmod`.
3) Use the major number in this command:

    ```
    [target]# mknod /dev/module c 254 0
    ```

    Skip this step if the node already exists.
4) Determine the various addresses from of the loaded module:

    ```
    [target]# grep '\.text' module.map
    ```

```
[target]# grep '\.rodata' module.map
[target]# grep '\.data' module.map
[target]# grep '\.sdata' module.map
[target]# grep '\.bss' module.map
[target]# grep '\.sbss' module.map
```

5) You can also just view the `module.map` file to find these addresses:

   Halt the target with the BDI2000: `BDI> halt`

6) Open the `module.map` file and select a suitable symbol to break on. Then set an initial breakpoint in the BDI2000: `BDI> bi <0xAddress>`

7) Give control back to your target system: `BDI> go`

8) Initiate activity to the device: `[target]# echo > /dev/module`

9) This issues a write if the driver supports it. Skip this step if you use another mechanism to initiate activity to the module.

10) The BDI2000 will display `entered debug mode`, the current PC should be at the break point. At this point clear the breakpoint we just set:

```
BDI>ci
```

11) Start GDB using the vmlinux file that has debug symbols:

```
[host]# <prefix-name>-gdb vmlinux
```

12) Add symbols into GDB (addresses are notional):

```
(gdb) add-symbol-file <path-to-module-dir>/ex_sw.o 0x c2000060\-s .rodata
0xcf030354\-s .data 0xcf030488\-s .sdata 0xcf030488\-s .bss 0xcf030519\-s .sbss
0xcf03051c 0xc2000060=text address
```

13) Connect GDB to the BDI2000

```
(gdb) target remote 192.168.0.131:2001
```

   where `192.168.0.131` is a notional IP address assigned to the BDI2000 and `2001` is the default port number.

14) Perform a list command,

```
(gdb) list
```

   Current source code will be displayed.

   Perform a few steps in GDB, you should see a step occur in the BDI2000:

```
(gdb) stepi
(gdb) stepi
```

15) Finally, set your own break points in GDB and see if it the BDI2000 hits the breakpoints:

```
(gdb) b (breakpoint-1)
(gdb) b (breakpoint-2)
(gdb) c
```

If upon doing the `insmod` command the module gets loaded and runs instantly, then you will not get a chance to set a breakpoint with the BDI2000. For this case you will have to change the procedure for setting a breakpoint and setting up the environment for debugging. One alternative is to put a conditional loop in the module at main where the loop will send you if you modify a register. Once you are in the loop you can halt the system through the BDI2000 and set the register to continue executing out of the loop. At this point you should proceed to issue the command `insmod` and make the `module.map` file, then restart the system and set a breakpoint in the BDI2000 before the `insmod` command is used again. This breakpoint is at a location in the `module.map` file that was made previously and should match the new `insmod`. Once this breakpoint is set, load the module as normal with `insmod` and the BDI2000 will indicate that the breakpoint has been hit.

APPLICATION DEBUGGING

The BDI2000 cannot be used to debug Linux applications; it is designed so that it can access kernel space memory, but it has no information about user space memory. Below is a diagram giving you a basic idea of how Linux divides the virtual address memory locations. Upon startup with `startlinux`, the Micromonitor boot loader moves the kernel from the Tiny File System (TFS) into memory and jumps to the kernel start routine. Linux then takes over and brings up the kernel. Once Linux is running, it enables the MMU and uses virtual addressing. The Linux kernel occupies a quarter of the virtual address space starting from address `0xC0000000`; this region is also known as “Kernel Space”. Linux also creates an “Application Space” where it loads user-land programs. The virtual address for application space is different for different implementations of Linux; it can be at `0x02` or wherever the developer assigns it to be. Linux assigns the address to the program dynamically when it is loaded and has no fixed location where Linux will load a particular application. You will need to look at the mapfile under the PID entry for your program in the /proc file system to determine its memory map.

System Calls is the API to interface to IO and kernel space. Kernel Space is defined as all virtual addresses starting with `0xC`. Restricted access, applications must use system calls to access kernel Space. The application normally occupies a certain amount of memory and variables in the application do not get assigned their own memory space until an access is made to them; hence, there is no guarantee that a virtual address actually translates to a physical address. The BDI2000 is not designed to account for this type of behavior. Furthermore, data can get swapped out from application memory to save memory space, and gets loaded to a different location on the next access. This moves the breakpoint location, and this can confuse the BDI2000 when it tries to hit a breakpoint. In summary, the BDI2000 cannot be used for application debugging. You can, however, perform application debugging by using the gdbserver that resides on your target Linux machine. The connection steps are similar to what you do when you connect to the BDI2000. You have to make sure that the executable you use to debug on your host has debug symbols built-in. You cannot use the same executable that gets loaded on your target because it is not built with the proper debug headers. Read the steps describing application debugging below.

PROCEDURE FOR APPLICATION DEBUGGING

1.) On your target launch the application with the gdbserver specifying the host IP:

```
[target-session]# gdbserver 192.168.0.210:2001 <target-app>
```

2.) On your host start GDB with the application to debug on the host:

`[host-session]#  arm-linux-gdb <target-app>` (or mips-linux-gdb)

3.) From GDB connect to the gdbserver on your target specifying the target IP:

`(gdb) target remote 192.168.0.131:2001` (notional settings)

4.) You can now set breakpoints and begin debugging.

```
(gdb) b main
(gdb) cont
```

Debug as usual with GDB commands from this point forward.

7.0 Networking

7.1 Default Kernel Configuration with Networking Enabled

The GX-Linux BSP ships with a default kernel configuration that has networking enabled. With the supplied default configuration, you will only need to use the uMON to set networking values appropriate for your network. After connecting the target hardware to a LAN and booting the kernel, you should be able to use the network. The uMON passes the network settings to the Kernel via the `startlinux` script.

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

```
# ifconfig   or /sbin/ifconfig
# route
# ping <known-network-IP-address>
# ...etc...
```

7.2 Network Configuration

The network is automatically configured to mimic the settings in uMON when GX-Linux boots; however, for modification or adding a second Ethernet configuration (for those targets with dual Ethernet), follow these simple set of instructions.

The Ethernet driver and board configuration are setup for `eth0` at install time. For boards with dual Ethernet, the board is labeled `ENET0` and `ENTET1`, where `ENET0` is mapped to `eth0` and `ENET1` to `eth1`. After Linux bootup, to enable a second ethernet device, you can manually enter the following commands in the serial terminal window:

```
# ifconfig eth1 <ip-address> netmask <netmask> up
```

Example

```
# ifconfig eth1 192.168.0.111 netmask 255.255.255.0 up
```

To Disable either `eth0` or `eth1`, do the following:

```
# ifconfig eth1 down
```

or

```
# ifconfig eth0 down
```

You can re-enable either port if no reboot has occurred by issuing:

```
# ifconfig eth1 up
```

or

```
# ifconfig eth0 up
```

Issuing `ifconfig` shows the status of what is up and down with the dual Ethernet devices. Note: MAC addresses are assigned from what is set from the factory, and they are labeled on the uSBC module. If the uSBC supports two Ethernet MACs, they will be assigned one LSB difference. For example:

```
00:23:31:66:00:20  for eth0
00:23:31:66:00:21  for eth1
```

The user may add a command sequence manually or add it to the `/etc/init.d/rcS` file in rootfs and varetc to be executed automatically at startup. The `rcS` file is like an `autoexec.bat` file in DOS that executes commands as given at startup. Rebuild and load the romfs image to take the changes to `rcS`.

Example from the Build Directory:

```
# make romfs
# ttftp <target-ip> put romfs.img romfs.img
```

8.0 Serial Port

8.1 Serial Devices

8.1.1 Introduction

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

- `/dev/ttyS0`
- `/dev/ttyS1`
- `/dev/console`

8.1.2 Default Serial Driver Configuration

The GX-Linux BSP ships with a default kernel configuration that has the appropriate serial driver enabled, as shown in the table below:

- `/dev/ttyS0` (Used for console)
- `/dev/ttyS1` (Available)
- `/dev/console` (Mapped to /dev/ttyS0)

8.2 Program Access to `/dev/ttyS0`

With 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 make xconfig setting has the effect of enabling the `CONFIG_AT91RM9200_SERIAL` definition used during the build of the kernel. If you hardware supports `/dev/ttyS1`, this will enable the second serial port as well.

8.3 System Console

8.3.1 Default Console

The `/dev/console` is the device underlying system's `stdin`, `stdout`, and `stderr` file descriptors. In the GX-Linux BSP, the kernel first tries to open `/dev/console` when it boots to serve as the console. Failing that, it will try to open `/dev/ttyS0`.

8.3.2 Login via the Serial Port

If you attach a terminal or a computer running terminal emulation software to the target hardware's serial port, then you can login to the target hardware via the serial port.

8.3.3 Standard IO

Once logged in over the serial port, the system console is automatically mapped to the `/dev/ttyS0` device. This 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.

8.3.4 Restrictions

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`.

8.4 Serial Settings

Default Baud Rate

By default, the target hardware `/dev/ttyS0` serial port starts up at 38400 baud and `/dev/ttyS1` serial port, if available, or starts up at 9600 baud (most Linux workstation's start up at 9600 baud). The `/dev/ttyS0` baud rate value of 38400 was chosen to be compatible with the uMON target hardware bootloader. 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 `stty` utility or `ioctl()` calls on `/dev/ttyS0` or `/dev/ttyS1`, as appropriate, for changing serial port speed at runtime.

The default `/dev/ttyS0` settings are:

- `38400 baud`
- `8 data bits`
- `no parity`
- `1 stop bit`

The default `/dev/ttyS1` settings are:

- `38400 baud`
- `8 data bits`
- `no parity`
- `1 stop bit`

8.5 Kernel Console over Serial

Default Configuration

In the `make xconfig`, you will encounter the option `Console on Amtel AT91RM9200 serial port`. This 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()`:

- `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.

The setting we use here only applies to the kernel `printk()` messages.

Section 9.0 How to Use GNU X-Tools

9.1 Introduction

This section is intended to provide an abbreviated overview of using the GNU compiler and tools to generate executable programs for a target processor. Mastering the toolsuite and using it to develop embedded systems on a practical scale requires greater in-depth knowledge than can be covered here. If you are considering the GNU cross-tools for full-scale development, it is suggested that you acquire the manuals listed in the bibliography (see Bibliography). The ones having unrestricted reproduction rights are included in the docs directory on the GNU X-Tools CD-ROM. The O'Reilly book, *Programming with GNU Software*, by Mike Loukides and Andy Oram, is highly recommended, being complementary to this User Guide as well as a comprehensive guide to the GNU tools. The remaining texts provide more in-depth knowledge for specific toolsuite components as needed. In addition, Microcross offers a *GNU X-Tools Training Guide* as a complimentary product. It is based on an on-site three-day training course for beginning to intermediate level audiences. The course material is tailored to the GNU X-Tools for embedded development, and the format is geared to a self-paced training program.

9.2 Using the GNU X-Tools Command Line Tool `(xtools)`

The GNU X-Tools command line tool is a shell tool designed to provide several functions that facilitate convenient management of the GNU X-Tools cross-toolsuites from a command shell environment. For many developers, this method is more convenient than using a GUI. The following functions may be invoked from any `Bash` Shell prompt:

1) `xtools` -- Execution of the `xtools` command without arguments will show an abbreviated help screen that summarizes the command options described in this section. In addition, execution of an incomplete command (insufficient or incorrect arguments) will produce a help dialog for that command.

2) `xtools <target-alias>` -- This command starts a new bash shell having the appropriate environment variables and aliases initialized such that the cross-toolsuite components for the designated target are invoked by the familiar command names, in lieu of the native tool chain elements. For example, after starting a shell using the command line `xtools arm-elf`, the command `GCC` would actually start the compiler `arm-elf-gcc`. This facility allows the command line user to execute the cross-tools using the same command names as used in the native shell environment. This remapping of tool command names remains in effect until the shell is exited via `Ctrl-D` or the `exit` command. The GX-Linux source tree is configured for cross-compilation, so the command is not required for any builds or makefile targets.

3) `xtools status [path prefix]` -- This command generates a summary screen which displays the installation status of each GNU X-Tools cross-toolsuite in terms of the types, counts, and locations of files associated with each toolsuite. The command displays the toolsuite status by target-alias vs. the population of files in the following format:

```
Column 0 Target Name (target-alias) for this toolsuite
Column 1 (/usr/bin/<target-alias-*) files
Column 2 (/usr/<target-alias>/bin/*) files
Column 3 (/usr/<target-alias>/lib/*) files
Column 4 (/usr/<target-alias>/include/*) files
Column 5 (/usr/lib/gcc-lib/<target-alias>/*) files
Column 6 (/usr/man/man1/<target-alias>-*) files
Column 7 Total file count for this toolsuite
```

The optional path prefix argument allows the utility to examine the installation state in which the user may have elected to manually install, or build/install toolsuite(s) into an alternate location (other than `/usr`).

4) `xtools install <target-alias> [archive-path]` -- This command will install a complete cross-toolsuite from the distribution tarball or zipset located on the specified archive path. All files are unarchived to the proper location (see above) regardless of the current directory state, and subsequent invocation of a bash shell using the `xtools <target-alias>` command will render the toolsuite ready to use. If the archive path is not provided, a default path is selected. The command `xtools install` will display the default path, which is `/mnt/cdrom` unless it has been changed or overridden by the global environment variable`$XTOOLS_ARC_PATH`.

5) `xtools remove <target-alias> [path prefix]` -- This command allows the user to selectively remove a designated toolsuite from the system. In the unusual case where the toolsuite has been manually installed to some root prefix other than `/usr`, the second option can be used to specify the root prefix (such as `/usr/bin`) from which removal of all toolsuite files is desired. This command is most useful if you have performed a build from sources and subsequent install to an alternate root prefix path, and now desire to remove the installed tools.

6) `xtools remove-all go [path prefix]` -- The `remove-all` command allows the user to summarily remove all toolsuites from the system. With the exception of the files `/usr/bin/xtools`, `/usr/bin/xtools.rc`, and `/etc/profile` (Cygwin only); this command affects a complete removal of GNU X-Tools toolsuite binaries from your system.

7) `xtools install-libs <target-alias> [arcpath]` -- This option allows an administrator to update or reinstall toolsuite libraries only. This feature is useful for selectively reinstalling libraries that have been over-written, or updating the libraries from a newer toolsuite version without updating the entire toolsuite. It updates the following files:

```
/usr/<target-alias>/lib/*/libc.a
/usr/<target-alias>/lib/*/libstdc++.a
```

It extracts the library files from a Microcross GNU X-Tools Toolsuite distribution CD-ROM, or filesystem image. The optional `[arcpath]` parameter may be used to locate the update files. **Note:** This feature is currently experimental and may change.

8) `xtools install-hdrs <target-alias> [arcpath]` – This option, like option 7, allows incremental updates to an installed toolsuite except that it updates the library headers files. It updates the following files:

```
/usr/<target-alias>/include/*/*.h
```

9.3 Invoking GNU X-Tools

To invoke GNU X-Tools from the command line in the shell, type `xtools target-alias` and from that point on you do not need to type the alias at any point further. The target-alias will become part of the command line prefix (i.e., `target-alias$`). To change or exit the target-alias mode, type `exit` at the command line.

From Command Shell, type: `$ xtools arm-linux`
The prompt will now show: `arm-linux$`

Issuing the following commands can be accomplished using the shortened version of tool name (i.e., `gcc`, `as`, `ar`, etc.), provided that you perform the `xtools <target-alias>` command as shown above. All GX-Linux sources are pre-configured for cross-compilation, so the xtools script is not necessary.

- *target-alias-***gcc**
  Invokes all the necessary GNU compiler passes for the specific target processor toolsuite using (i.e., arm-elf, mips-elf, ppc-elf, etc.).

- *target-alias*-**cpp**
  Invokes the preprocessor that processes all of the header files and macros that the target requires.
- *target-alias*-**gcc**
  Invokes the C compiler that produces assembly language code from the processed C files.
- *target*-alias-**g++**
  Invokes the C++ compiler that produces assembly language code from the processed C++ files.
- *target*-alias-**gdb**
  Invokes the GNU Debugger with command line input (see Debugging with GDB in the PDF files on the GNU X-Tools CD.
- *target*-alias-**gdbtk**
  Invokes the GNU Debugger with a visual GUI interface called Visual GDB by Microcross (a.k.a. Insight).
- *target-alias*-**as**
  Invokes the GNU assembler that produces binary code from the assembly language code and puts it into an object file.
- *target-alias*-**ld**
  Invokes the linker that binds the code to addresses links the startup file and libraries to the object file and produces the executable binary image.

9.4 Using the GNU X-Tools Toolsuite

The first step to developing code with the GNU X-Tools toolsuite is creating and editing the source code. Microcross provides the Visual X-Tools IDE as companion product for code editing and project management; however, a programmer can use any text line editor to create source files and makefiles and use the GNU X-Tools as the production build environment. Free editors under Cygwin include VIM, an improved VI editor clone, and Xemacs, a powerful editor and project manager. Under Linux, the user also has a wide variety of choices including the Microcross Visual X-Tools IDE and GNU Xemacs edit (see docs in the Linux distribution).

9.4.1 Simple Example

Start a shell (xterm on Linux and GNU X-Tools Shell on Windows/Cygwin) and issue the command `xtools <target-alias>`, where the target alias is selected from the toolsuite you are using (i.e., arm-linux, mips-linux, mipsel-linux, ppc-linux, etc.), depending on the toolsuite you have installed.

The following simple example shows you quickly how easy it is to get a program compiled, linked and executed in an Instruction Set Simulator (ISS) and debug it using Visual GDB.

```
$ xtools <target-alias>
```

In our example, we use the arm-linux.

```
$ xtools arm-linux
arm-linux$ cd /home/test
```

In the last step, we assumed that the Microcross test directory either was installed in Cygwin's `/home/test` or Linux's `/home/test`.

```
arm-linux$ gcc –g –o pascal.x pascal.c
```

The program was compiled and linked in one step. You will need to add this file to the Linux filesystem and run it on the target processor using the procedure in Section 3. To debug this simple example, follow the procedure in Section 6.

This concludes the simple example showing how easy it is to compile and link a program with GNU X-Tools. The GNU X-Tools toolsuite is very powerful and can build any conceivable program a developer can code. Now, we need to go into more detail on how the GNU X-Tools work.

9.4.2 GNU X-Tools Toolsuite Description

The `GCC` program is actually a control program that executes the compiler components to produce the desired output, which is usually a compiled and linked executable program image. By manipulating the many `GCC` options and controlling the input file types, the functions of `GCC` are greatly expanded. The `GCC`, however, is but a single component. It is actually a control program that calls other components that perform separate steps to create an executable binary. The components are described as follows and show graphically in Figure 9.1:

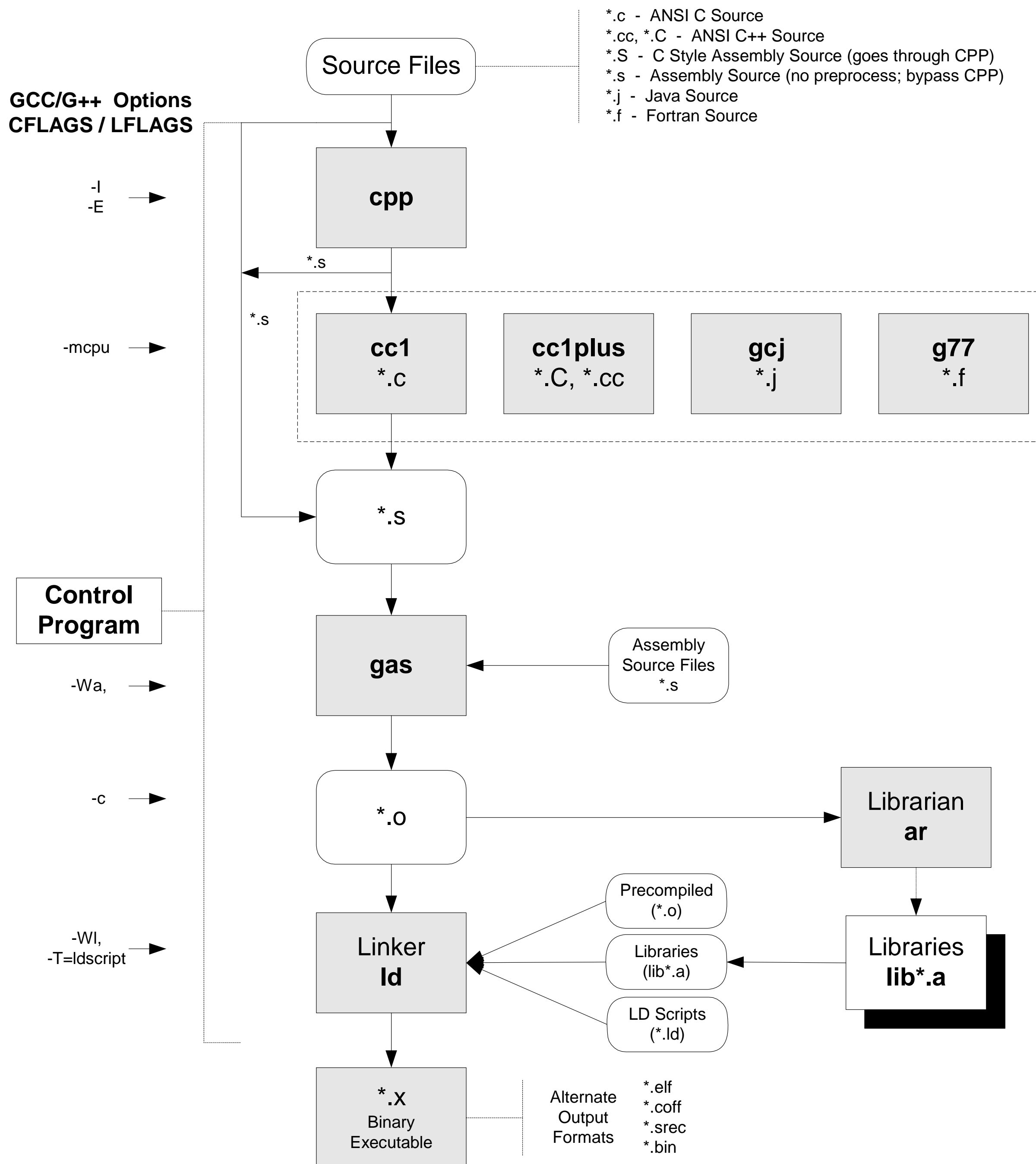
preprocessor
Performed by `cpp`, which is invoked by `GCC`, the preprocessor resolves directives like `#define`, `#include`, and `#if`. Preprocessing establishes the type of source code to process.

compiler
Performed by `GCC`, the compiler pass, which produces assembly language from the input files, and passes the assembly source directly to the assembler phase.

assembler
Performed from `GCC` by `as`, the GNU assembler. It takes the assembly language as input and produces object files with `.o` extensions. The assembler output is relocatable object code `(.o)`.

linker
Performed by `ld`, the GNU linker. Linking completes the compilation process, combining all object files (newly compiled, and those specified as input) into an executable file. This step completes the final stage, where the `.o` modules are placed in their proper places in the executable file. Library functions that the program refers to are also placed in the file. `GCC` performs this task by internally invoking the linker. `GCC` also cleans up by deleting any object files that it created from source files; however, it does not cleanup any pre-existing object files that you specified on the command line. `GCC` normally invokes all of these compilation steps when converting a C source program into an executable. By using command line options for `GCC`, these steps may be invoked separately or in some combinations. This provides some flexibility when building large programs, or using assembly language sources, or debugging.

Figure 9.1. GNU X-Tools Flow Diagram



9.4.3 Control Program (GCC)

Exactly how the GCC processes any file depends on the file's name. The control program strips the initial part of the name, and then determines how to process the file on the basis of the filename's extension. In each case, the control program passes the file to the appropriate program for preprocessing, compilation, or assembly, and it links

all resulting object modules together to produce an executable file. The following table 9.1 shows how the control program recognizes different file types:

Table 9.1. Controlling Compilation / Linking

Input File	Interpretation	Action
file.c	C source*	Preprocessed and compiled by GCC
file.C	C++ source **	Preprocessed by GCC and compiled by g++
file.cpp	C++ source *	Preprocessed by GCC and compiled by g++
file.CPP	C++ source **	Preprocessed by GCC and compiled by g++
file.cc	C++ source *	Preprocessed by GCC and compiled by g++
file.cxx	C++ source *	Preprocessed by GCC and compiled by g++
file.h	C or C++ header file	Precompiled header
file.s	Assembly language source*	Assembly by as
file.S	Assembly language source**	Preprocessed and assembled by as
file.o	Compiled object module*	Passed to ld
file.a	Object module library*	Passed to ld

* Lower case file extension.
** Upper case file extension.

All other files, together with options that GCC does not recognize, are passed to ld, the linker either the native linker supplied by your vendor, or the GNU linker. As a result, almost all linker options are available directly through GCC.

In the commands shown earlier in this Section, the –l and –L options were actually passed to the linker. In general, GCC passes on unrecognized options to the linker, so you can specify linker options without having to invoke the linker separately.

If you're writing C++ code, you can use the G++ command instead of GCC. You're actually getting the same compiler; however, when it is invoked as G++, the compiler expects C++ instead of vanilla C source code. In addition, G++ uses different default libraries.

If you want either to preserve the output of GCC at some intermediate stage for debugging purposes or to manipulate the code directly, you can do so. Here is how each stage works:

preprocessing

A –E option in GCC sends the preprocessed code to the standard output, instead of compiling the program.

compilation

To save the assembly language output, run GCC with the –S option. This produces files whose names end with .s in place of the source file's .c.

assembly

As we have seen, running with –c produces object files whose names end with .o.

We have just finished a long discussion of the many kinds of input that the compiler takes and the different kinds of output that it can provide. Pictures are not always worth a thousand words, particularly when it comes to summarizing a lot of disparate information. But it may help you to remember this information if you view the compiler as a kind of "machine" with different inputs and outputs, as outlined earlier.

The input file's name determines where it goes into the machine. The `.c`, `.S`, `.cc`, `.cpp` and `.C` files go to the preprocessor, `.i` and `.ii` files go straight to the compiler, and so on. Compilation options determine which stage of the machine produces output (i.e., how many stages of the machine you run). The `–S` means that you stop after the assembler and the output filename ends with `.s`. If you keep this picture in mind, the compiler's machinations will not seem so strange; you will stop seeing preprocessing, compilation, assembly, and linking as separate steps and come to see compilation as one big assembly line, for which `GCC` is the production manager.

The following tools listed in Tables 9.2, 9.3, and 9.4 are the main tools for developing projects with a GNU X-Tools Toolsuite.

Table 9.2. GNU X-Tools Compiler and Development Tools

Tool	Description
`as`	GNU assembler
`cpp`	C preprocessor
`gcc`	Optimizing ANSI compliant GNU Compiler Collection (controller of all compilation)
`gdb`	GNU debugger for source and assembly debugging with command line
`gdbtk`	Debugger using a graphical user interface called Visual GDB (a.k.a. Insight)
`g++`	Optimizing ISO tracking GNU C++ compiler
`gasp`	GNU assembler preprocessor
`ld`	GNU Linker

Table 9.3. GNU X-Tools Libraries

Tool	Description
`glibc`	ANSI C runtime library
`libgloss`	Support library for embedded targets (*board support for cross-development*); (deprecated) will be replaced by a future product
`libm`	C math subroutine library
`libstdc++`	GPL C++ class library, implementing the ISO 14882 Standard C++ library

Table 9.4. GNU X-Tools Binary Utilities

Tool	Utility Description	Target Dependent
`addr2line`	Converts addresses into file names and line numbers	yes
`ar`	Creates, modifies and extracts from object code archives	yes

`diff`
`diff3`
`Sdiff`
	Comparison tools for text files	no

`make`	Compilation control program	no
`nm`	Lists symbols from object files	yes
`objcopy`	Copies and translates object files	yes
`objdump`	Displays information from object files	yes
`patch`	Installation tool for source fixes	no
`ranlib`	Generates index to archive contents	yes
`readelf`	Displays information about ELF format object files	no
`run`	Standalone simulator	yes
`size`	Lists file section sizes and total sizes	yes
`strings`	Lists printable strings from files	yes
`strip`	Discards symbols	yes

9.5 Controlling the Tools Using the GCC

The GCC (GNU Compiler Collection) control program can be run like the following, producing an executable file from a number of C or C++ source files. Both examples assume that the target toolsuite is installed prior to executing the build commands. The examples shown are with GCC, which controls the compilation of C, C++, and assembly files. Also, the user must name the C files with the .c extension and C++ files with .C , .cpp or .cc extensions.

```
$ target-alias-gcc –options –o <program.out.name> <first.c> <second.c> <third.c>
<fourth.c> …
```

Or

```
$ xtools <target-alias>
<target-alias>$ gcc –options –o <program.out.name> <first.c> <second.c> …
```

Or

```
$ xtools <target-alias>
<target-alias>$ gcc - options one_or_more_source_files.c -o program.out.name
```

Or

```
<target-alias>$ gcc - options -c source_file.c
<target-alias>$ gcc - options file1.o . . . fileN.o -o program
```

All alternatives actually look the same to the compiler parts. The program executed under the name of GCC is just a front that handles options and temporary files and calls the real compiler parts: cpp, the C preprocessor. It takes care of preprocessor directives, such as include files and macro expansions. It also removes comments. The result is a file with the C code, lots of white space and some line-numbering directives that the compiler core can use in warning and error messages.

The –o filter argument tells GCC to name the executable file program. If you don't specify an –o argument, GCC chooses the default name a.out, which is not particularly informative (and would cause multiple executables to overwrite each other). So, most programmers use the –o argument to name the program.

Since the inputs to the last example above are all object files, no compilation or assembly is required: GCC always invokes the linker. Using GCC to invoke the linker is preferable to using ld separately, because GCC ensures that the program is linked with the correct libraries and initialization routines.

9.5.1 GCC Options Commonly Used

–c

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

-o *file*

The –o and not –c tells GCC to create a linked executable with a user defined filename as its output file. Not specifying –c or –o will tell GCC to create a default executable named a.out for all cross-compilers and a.exe for the native Cygwin compiler.

-D

One, the `-D` option, acts like `#define` in the source code: it sets the value of a symbol.

```
<target-alias>$  gcc -c -D first=\"info\" -D second example.c
```

The first `-D` option sets `first` to the string `info\` (because of the backslashes, the quotation marks actually become part of the symbol's definition). This can be useful for controlling which file a program opens. The second `-D` option defines the `second` symbol. It happens to set it to the value 1, the default, but you probably don't care; your program just uses a `#ifdef` directive to check whether it's set.

-E

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files which don't require preprocessing are ignored.

-S

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix `.c`, `.i`, etc., with `.s`.

Input files that don't require compilation are ignored.

-v

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-###

Like -v except the commands are not executed and all command arguments are quoted. This is useful for shell scripts to capture the driver-generated command lines.

-pipe

Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

--help

Print (on the standard output) a description of the command line options understood by gcc. If the `-v` option is also specified then `—help` will also be passed on to the various processes invoked by gcc, so that they can display the command line options they accept. If the `-Wextra` option is also specified then command line options which have no documentation associated with them will also be displayed.

--target-help

Print (on the standard output) a description of target specific command line options for each tool.

--version

Display the version number and copyrights of the invoked GCC.

-Uname

Some symbols – usually those identifying the type of computer system you're compiling on – are automatically defined by the compiler. If you want to suppress one of these symbols, use the `–Uname` option. This is equivalent to putting `#undef name` at the beginning of each source file.

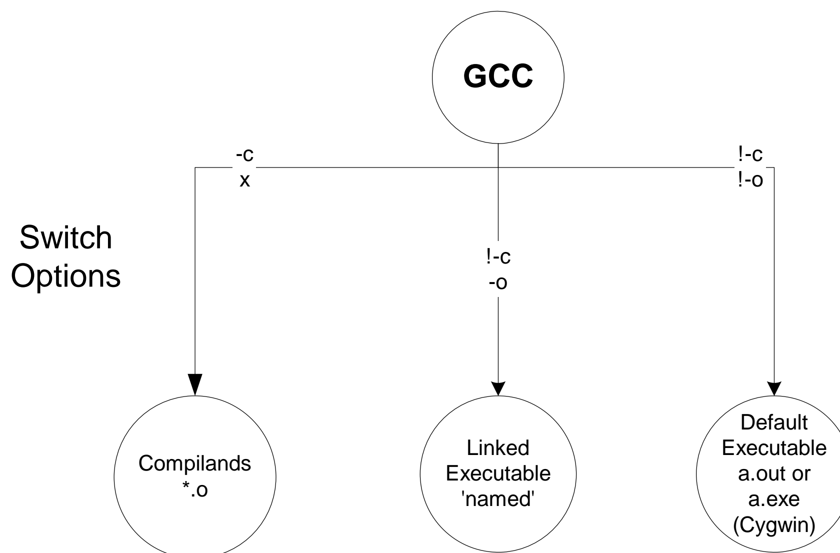
`-I`

If you have an include file in a non-standard directory, specify this directory in a `–I` option. For example, suppose you have two directories for code, `/usr/src` for source files and `/usr/headers` for header files. While you're compiling in the `/usr/src` directory, you can tell `GCC` where to find the header files through the command:

```
<target-alias>$ gcc –I../headers example.c
```

Figure 9.2 below summarizes the three primary modes of `GCC` with switch actions.

Figure 9.2. Three Primary Modes of GCC



9.6 Controlling Linker from GCC

Another common compilation option is `–l`, which specifies a library and `L`, which specifies the path to a non-standard library. Important, `–l` must be specified at the end of the command after the source/object filenames, while all other options are specified before the source/object filenames. The examples shown are with `GCC`, but the user can substitute `g++` in place of `gcc` along with the appropriate C++ files and libraries needing linked. Here's a typical command; the inputs are object files, so the command just runs the linker.

```
<target-alias>$ gcc –o example –L/src/local/lib main.o first.o –lm
```

This command tells `GCC` to look for libraries first in `/src/local/lib path`, then in the standard location. If someone has put a local version of a library in `/src/local/lib`, that version is used in preference to the standard version in `/usr/lib`. The `–lm` specifies the library name -- actual library name is `libm.a`. The prefix and suffix to the library name can be dropped on all library names. The naming convention for GNU libraries is lib<name-of-lib>.a (static libs) and lib-<name-of-lib>.so (shared libs).

Unix / Linux linkers search libraries in the order in which they occur on the command line and only resolve the references that are outstanding at the time the library is searched. Therefore, the order of libraries and object

modules on the command line can be critical (this is why -l options come after the filenames). Consider the command:

```
<target-alias>$  gcc -l mine first.c
```

This command searches for the library file lib**mine**.a to resolve any function references needed for linking; however, the linker has not yet processed the object module for file4.o (created by the GCC command and normally deleted if compilation and linking are successful). Therefore, there are no outstanding function references, and the library search has no effect. If the program needs this library, the GCC command produces Undefined symbol messages during the loading phase, and the linker does not produce an executable file. To perform this compilation correctly, enter the command:

```
<target-alias>$  gcc first.c -lmine
```

Now the loader searches the library after processing first.c and is able to resolve any references requiring this library.

When compiling a C program, you often don't need to list any libraries explicitly on the command line. The GCC automatically searches the system call library, I/O library, and run-time initialization routines. If you use any math routines, you'll need to search for the math library (-lm); if you're compiling C++ code, you need to include the C++ libraries (-lstdc++).

9.7 Compilation Options

The following sections discuss other important options that are available with the GCC. There are many, many other options – perhaps several hundred all-together – that control various details of compilation and optimization. The chances are that you will never need these, but it won't hurt to familiarize yourself with the complete documentation for GCC. Complete coverage of GCC options is provided in *Using and Porting GNU CC* [reference 20].

9.7.1 Displaying compiler behavior

The -v (verbose) option prints the compiler's version number and complete details about how each pass is executed. This option is particularly useful for finding out exactly which options your program is being linked with.

9.7.2 C Language Options

We'll start by listing a few options for controlling the warning messages that GCC produces. There are many options for controlling warnings; it's possible to request (or inhibit) many warning messages on a per-message basis. We're not sure that's really useful; we'll limit ourselves to a few options that control large group of messages:

-w
Suppress all warning messages.

-W
Produce some additional warning messages about legal (but questionable) coding practices.

-Wall
Produce even more warning messages about questionable coding practices.

-Wtraditional

Produce warning messages about code that is legal in both the Kernighan and Ritchie and ANSI definitions of the C language, but behaves differently in each case.

`-Werror`

Make all warnings into errors; that is, don't attempt to produce an object file if a warning has occurred.

Now, we'll discuss how to control various features of the C and C++ languages. There are basically three options to worry about: `-traditional`, `-ansi`, and `-pedantic`. In most cases, it's fairly easy to tell which you want. Older C code – code that pedants the standard – should be compiled with `–traditional`. Newer code that has been written to conform to the ANSI standard should be compiled with `–ansi`. Either option can accept prototypes, where you specify the arguments on the same line as the function name, as in `func (char *arg)`.

Note that the ANSI C standard and "traditional" (Kernighan and Ritchie) C both define the behavior of the preprocessor – either explicitly or implicitly. Therefore, the options listed below affect both `cpp` (C preprocessor) and `GCC:`

`-traditional`

Supports the traditional C language, including lots of questionable, but common, practices. The traditional option also supports all of the FSF's extensions to the C language.

`-ansi`

Supports the ANSI C standard, though somewhat loosely. The FSF's extensions are recognized, except for a few that are incompatible with the ANSI standard. Thus, ANSI programs compile correctly, but the compiler doesn't try too hard to reject non-conformant programs, or programs using non-ANSI features.

`-pedantic`

Issues all the warning messages that are required by the ANSI C standard. Forbids the use of all the FSF extensions to the C language and considers the use of such extensions errors. As the `GCC` manual points out, `-pedantic` is not a complete check for ANSI conformance, it only issues errors that are required by the ANSI standard.

9.7.3 Preprocessor Options

The following set of options control the `cpp` preprocessor from the command line:

`-M`

Read the source files; figure out which files they include, and output lists of dependences for make. There is one dependency list for each source file. The dependency lists are sent to standard output, and compilation doesn't proceed past preprocessing (i.e., `-M` implies `–E`). This option can make it much easier to generate correct makefiles.

`-C`

The preprocessor normally deletes all comments from the program. With `–C`, it doesn't. This flag may be useful in conjunction with `–E` when you are trying to make sure that the preprocessor is doing what you intended. In such cases, leaving you comments in may be handy. The `–C` option doesn't automatically imply `–E`, but `GCC` won't let you use `–C` on the command line unless `–E` is also present.

9.7.4 Options to Specify Libraries, Paths and Startup Files

The following options are common for embedded developers, but not in native environments:

`-nostartfiles`

Don't use the standard system startup files when linking. Normally the `crt0.o` file gets linked in as the standard start file; however, most embedded developers will need to replace the standard start file with a custom start file. When used with GCC, add the correct option `–Wl,-nostartfiles` and add the custom startfile with appropriate entry symbol to the linker script (e.g., `ENTRY(_start)`). To learn more about how to create linker scripts and startfiles, read the GNU X-Tools Training Guide or the Microcross Visual X-Tools User Guide.

`-nodefaultlibs`

Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files and used normally, unless `-nostartfiles` is used. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ANSI C) environments or to bcopy and bzero for BSD environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

`-nostdlib`

When linking, this option tells the linker do not use the standard libraries or startup files. This option is useful when you want to provide your own libraries, overriding the default libraries and use your own custom startup files. When used with GCC, add the correct option `–Wl,-nostdlib`. If you plan on augmenting the standard libraries with your own libraries, then do not use this option. Use a `GROUP` and `SEARCH_DIR` statement with appropriate arguments in the linker script – see the Visual X-Tools IDE User Guide or the GNU X-Tools Training Guide for examples on how to create linker scripts and startfiles.

`-nostdinc`

Do not search the standard system directories for header files. Only the directories you have specified with `-I` options and the current project directory are searched. By using both `-nostdinc` and `-I-` you can limit the include file search path to only those directories you specify explicitly.

`-static`

Link only to static libraries, not shared libraries. When used with GCC, add the correct option `–Wl,-static`. In GNU X-Tools, static link is the default setting.

`-shared`

If shared libraries are available, use them wherever possible, rather than static libraries. When used with GCC, add the correct option `–Wl,-shared`.

`-I`*`dir`*

Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I` option, the directories are scanned in left-to-right order; the standard system directories come after.

`-I-`

Any directories you specify with `-I` options before the `-I-` option are searched only for the case of `#include "file"`; they are not searched for `#include <file>`. If additional directories are specified with `-I` options after `-I-`, these directories are searched for all `#include` directives. Ordinarily all `-I` directories are used this way. In addition, the `-I-` option inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include "file"`. There is no way to override this effect of `-I-`. With `-I.` you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. `-I-` does not inhibit the use of the standard system directories for header files. Thus, `-I-` and `-nostdinc` are independent.

-L*dir*

Add directory *dir* to the list of directories to be searched for `-l`.

-B*prefix*

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself. The compiler driver program runs one or more of the subprograms `cpp`, `cc1`, `as`, and `ld`. It tries prefix as a prefix for each program it tries to run, both with and without `machine/version/`. For each subprogram to be run, the compiler driver first tries the `-B` prefix, if any. If the name is not found, or if `-B` was not specified, the driver tries two standard prefixes, which are `/usr/lib/gcc-lib/` and `/usr/local/lib/gcc-lib`. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your `PATH` environment variable. `-B` prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `-L` options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into `-isystem` options for the preprocessor. In this case, the compiler appends include to the prefix. The run-time support library file `libgcc.a` can also be searched for using the `-B` prefix, if needed.

-specs=*file*

Process *file* after the compiler reads in the standard `specs` file, in order to override the defaults that the `gcc` driver program uses when determining what switches to pass to `cc1, cc1plus, as, ld`, etc. More than one `-specs=` file can be specified on the command line, and they are processed in order, from left to right.

9.7.5 Debugging and Profiling Options

These options request the compiler to create additional code and an expanded symbol table for the various profilers and debuggers (`dbx`, `prof`, `gprof`, and the branch count profiler). They are extremely helpful for debugging and tuning code under development, but should not be used for production release versions of your program.

-p

Link the program for profiling with `prof`. When you execute a program compiled with this option, it produces a file named `mon.out` that contains program execution statistics. The profiler `prof` reads this file and produces a table describing your program's execution.

-pg

Link the program for profiling with `gprof`. Executing a program compiled with this option produces a file named `gmon.out` that includes execution statistics. The profiler `gprof` reads this file and produces detailed information about your program's execution. For example, the following command compiles the file `program.f`, generating code for profiling with `gprof`:

```
<target-alias>$ gcc -pg program.f
```

-g

Generate a symbol table for debugging. The `-g` option enables debugging with the GNU debugger, `GDB` or `Visual GDB`. The symbols used with standard GDB are called stabs. When compiling with Microcross' cross-compilers, set the `-g` option to produce the stabs symbols; you may use `-gstabs` as a verbose switch if you like. Other symbol options include `-gcoff, -gdwarf` and `-gdwarf2;` the `-gcoff` symbols are not supported in Microcross GNU cross-compilers; however, `-gdwarf` and `-gdwarf2` are supported in most target CPU toolsuites.

IMPORTANT NOTES: To perform assembly debugging of straight assembly language files (`.s` or `.S` files), you must use `-g` or `-gstabs` because the assembler will not generate any other symbol formats. The C/C++

compiler, however, will generate appropriate symbols for either -gdwarf or -gdwarf2 in addition to stabs. When compiling with any debug option, do not use optimization options. You cannot optimize code and debug it correctly using the GNU tools.

9.7.6 Optimization

GNU X-Tools GCC incorporates a sophisticated optimizing compiler; and for most target systems, it usually generates faster code than with the native compiler. Any -Ox optimization can be used with any -fx option (e.g., gcc -Os –funroll-loops –ffast-math –o outfile.x infile.c). Here are the most commonly used compilation options:

-O0

No optimization. This is the default. With optimization turned off, GCC tries to generate code that is easy to debug; you can set a breakpoint between any two statements and modify variables, and the program will behave exactly as it should. The GCC also tries to generate code quickly.

-O1

The compiler tries to moderately to reduce both the size of the compiled code and the execution time. Compilation is slower than with -O0 and requires more memory.

-O2

Enables more optimizations than -O1. Compilation time is even slower; the resulting code should be even smaller and faster.

-O3

Enables more optimizations than -O2. Try hardest of all to produce fast assembly code. Note that the emphasis is on fast: the resulting code may take much more room in memory because certain functions may be placed in-line and loops may be unrolled (as if each iteration in the for loop were written out independently).

-Os

Try to produce code that is small. The emphasis is on size, not speed. This option uses many of the same optimization algorithms as -O2, but with a different emphasis.

-ffast-math

Make floating-point arithmetic optimizations that violate the ANSI or IEEE standards. Code compiled with this option may yield incorrect results, but it will be slightly faster. Be sure to test your program thoroughly.

-finline-functions

Expand all “simple functions” into their callers. The compiler gets to decide whether any function is “simple” or not. Inline expansion is a two-edged sword; it can make a program faster (by eliminating calling overhead) or slower (by making instruction cache performance worse).

-fno-inline

Inhibit all inlining, even inlining that is requested by the inline keyword in the source code. The GCC performs inlining according to statements in the source code with both -O1 and -O2; the keyword is ignored if optimization is not in effect.

-funroll-loops

On some CPUs, loop unrolling can be a very important optimization. It minimizes loop overhead and creates many opportunities for further optimizations. With the -funroll-loops option, GCC unrolls all loops that have a fixed

iteration count known at the time of compilation. Loop unrolling is another risky optimization; it can improve performance, but it can also penalize performance significantly: the object file becomes larger, which can significantly hurt cache performance, and compile time also increases.

9.7.7 Passing Options to the Assembler or Linker

The `GCC` allows you to pass options directly to the assembler or linker when they are involved:

`-Wa,options`

Pass the `options` to the assembler

`-Wl,options`

Pass the `options` to the linker

In both cases, the `option-list` is just a list of options recognized by the assembler or the linker. There must not be any spaces in the list; options in the list are separated by commas.

Here is an example that is both instructive and useful: producing a listing of the assembly language generated, together with C source listings. To do this, we need to pass the `–alh` options to the assembler (generate listings of assembly code and high-level source); we also need to pass the `–L` option to the assembler (retain local labels). And we need `GCC's –g` option (generate additional symbols for debugging; the additional symbols tell the assembler where to find the source code). The resulting command looks like this:

```
<target-alias>$ gcc –c –g –Wa,-alh,-L source.c
```

Listings that include both assembly and source code are interesting from two standpoints. You may want to see how your code has been compiled; this is instructive, whether or not you are optimizing and even if you are not interested in assembly level debugging. What is more important, though, is that you can generate a C / assembly listing for optimized code. This can be very helpful for debugging under optimization. The big problem with debugging optimized code is that there is no longer a simple mapping from your source code into assembly language; therefore, the debugger cannot single step and perform symbolic debugging. With a listing, you can find out exactly what the compiler did to your code and get a much better idea of what the code is doing.

9.8 Using the GNU Assembler

The GNU assembler is really many assemblers folded into one (or many different programs with the same name, depending on how you look at it). You can usually ignore the assembler; the compiler invokes it automatically and is usually able to specify everything the assembler needs to know about your environment. In rare cases, you may need to ask for an assembly option explicitly; in these cases, you will need to run the assembler, `as,` a separate program or use `GCC's –Wa` option to pass additional options to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas. For example:

```
<target-alias>$ gcc -c -g -O -Wa,-alh,-L file.c
```

This above example emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this `-Wa` mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. You can call the GNU compiler driver with the `-v` option to see precisely what options it passes to each compilation pass, including the assembler.

Next, we discuss what the assembler does. We will not discuss the assembly language itself. The FSF's documentation explains the general syntax of assembly language, but refers you to the vendor's architecture manual for CPU dependent details: overall architecture, instruction set, etc..

The assembler takes a program written in an assembly language and produces an object module. By convention, assembly language programs have the extension *.s*. If no errors occur during assembly and if the object module contains no references to external (imported) symbols, the assembler makes the file executable and names it `a.out`. If the object module includes references to external symbols, `a.out` is not an executable. The linker is able to link this object module with other modules to produce an executable program.

To invoke the assembler, enter the command:

```
<target-alias>$ as -options <list-of-source-files>
```

Where `list-of-options` is a series of assembly options and `list-of-source-files` is one or more assembly language files (`.s`) -- `.S` (capital S) requires `GCC` to direct the source file to the preprocessor and then on to the assembler . Unlike most Unix assemblers, the GNU assembler can work on several files at a time.

The assembler has many options; most of them are architecture-specific and are used to describe the target processor more precisely. These options will be important to you if you are cross-compiling; check the FSF's manual for more details. Each appendix in this user guide briefly discusses the target dependent compiler and assembler options.

The following are a set of options and controls that are generally useful for invoking all GNU assemblers:

Common assembler switch options to all Targets

as	[-a[cdhlns][=file]] [-D] [--defsym sym=value]
	[-f] [--gstabs] [--help] [-I dir][-J][-K][-L]
	[--keep-locals] [-o objfile] [-R] [--statistics]
	[-v] [-version] [--version] [-W] [-w] [-x] [-Z]
	[-mbig-endian | -mlittle-endian] (if implemented)
	[-mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu] (if implemented)
	[-EB | -EL] (if implemented)
	[-O]
	[-O | -n | -N]
	[-b] [-no-relax]
	[-nocpp] [-G num] [-mcpu=CPU] (if implemented)
	[--trap] [--break]
	[--emulation=name]
	[-- | files...]

These assembler switch options are discussed in depth within the manual, *MC-Utilities.pdf* (docs directory on GNU X-Tools CD-ROM and in Cygwin/docs). In addition, each assembler has target specific switch options. The target-specific switch options are located in the Appendices of this User Guide.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the `.org` directive. These options enable listing output from the assembler. By itself, `-a` requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah` requests a high-level language listing, `-al` requests an output-program assembly listing, and `-as` requests a symbol table listing. High-level listings require that a compiler debugging option like `-g` be used, and that assembly listings (`-al`) be requested also.

Use the `-ac` option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing. Use the `-ad` option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list, .nolist, .psize, .eject, .title, and .sbttl`. The `-an` option turns off all forms processing. If you do not request listing output with one of the `-a` options, the listing-control directives have no effect. The letters after `-a` may be combined into one option, e.g., `-aln`.

-a[cdhlmns]

Turn on listings, in any of a variety of ways:

-ac	Omit false conditionals
-ad	Omit debugging directives
-ah	Include high-level source
-al	Include assembly
-am	Include macro expansions
-an	Omit forms processing
-as	Include symbols
=file	Set the name of the listing file

You may combine these options; for example, use `-aln` for assembly listing without forms processing. The `=file` option, if used, must be the last one. By itself, `-a` defaults to `-ahls`.

-D	**Ignored***
--defsym sym= value	Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading 0x indicates a hexadecimal value, and a leading 0 indicates an octal value.
-f	"Fast"—skip white space and comment preprocessing (assume source is compiler output). Warning: if you use `-f` when the files actually need to be preprocessed (if they contain comments, for example), `as` does not work correctly.
--gstabs	Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
--help	Print a summary of the command line options and exit.
-I *dir*	Add directory, `dir`, to the search list for `.include` directives. You may use -I as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, `as` searches any `-I` directories in the same order as they were specified (left to right) on the command line.
-J	Don't warn about signed overflow
-K	Issue warnings when difference tables altered for long displacements.
-L	Keep (in the symbol table) local symbols, starting with L.
--keep-locals	Keep (in the symbol table) local symbols. On traditional `a.out` systems these start with L, but different systems have different local label prefixes.
-M or --mri	This changes the syntax and pseudo-op handling of as to make it compatible with the ASM68K or the ASM960 (depending upon the configured target) assembler from Microtec Research.
--MD	`as` can generate a dependency file for the file it creates. This file consists of a single rule suitable for make describing the dependencies of the main source file.

-o	`objfile` Name the object-file output from as `objfile`. By default it has the name `a.out` (or `b.out`, for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.
-R	Fold the data section into the text section.
--statistics	Print the maximum space (in bytes) and total time (in seconds) used by assembly.
--strip-local-absolute	Remove local absolute symbols from the outgoing symbol table.
-v -version	Print the as version.
--version	Print the as version and exit.
-W	Suppress warning messages.
-w	Ignored*
-x	Ignored*
-z	Generate an object file even after errors.
-- | files...	Standard input, or source files (files ...) to assemble.

* This option is accepted for script compatibility with calls to other assemblers.

9.9 Using the Linker

The `ld` linker combines several object modules and libraries into a single executable file. It resolves references to external variables, external procedures, and libraries, creating a complete, self-sufficient program. You never need to invoke `ld` explicitly. In most cases, it is simpler to use the `GCC` command to link files, even if you do not have any source files to compile. The `GCC` guarantees that certain libraries will be present in the proper order even if they are not listed on the command line. If you use `ld` as a linker, you need to mention these libraries explicitly.

9.9.1 Invoking ld

The rules for invoking `ld`, if you must do so, are the same as for `GCC` or `as`. The basic `ld` command is as follows:

```
<target-alias>$ ld <list-of-options> <list-of-files-and-libraries>
```

Where `list-of-files-and-libraries` is a series of filenames and library specifications. To include a library in this list, use the notation `–lname`, where the name of the library file is either `/lib/libname.a` or `/usr/lib/libname.a`. The linker processes the `list-of-files-and-libraries` in order. When it reaches a library, it extracts only those modules that it currently needs to resolve external references. Consequently, the position in which libraries appear in this list is important. For example, the command:

```
<target-alias>$ ld prog1.o –lm prog2.o
```

results in an `Undefined symbol` message if `prog2.o` refers to any programs in the library `/usr/lib/libm.a` – unless you happen to be lucky and `prog2.o` only uses routines that the linker extracted for the sake of `prog1.o`. Note that libraries may refer to other libraries; thus, the command:

```
<target-alias>$ ld prog1.o –lat –lfo
```

leads to `Undefined symbol` messages if the `fo` library requires any routines from `at`.

This situation is more complex for a user-generated library. Such a library should contain an index, so that the linker can find each module regardless of its order within the library. Some systems always generate an index when you create or modify the library with the `ar` command (the GNU `ar` does this). On other systems you have to put in the index yourself by using the `ranlib` command.

If you want to create an executable file, the beginning of the first file in the list-of-files must be the program's entry point. This is not the same as the apparent entry point to your C source program. Before your program begins executing, the computer must execute a standard run-time initialization routine. To ensure that this is in place, `/lib/crt0.o` must be the first file in the `list-of-files-and-libraries`. This ensures that this initialization routine is linked to your program.

Alternatively, you can link by using the `GCC` command without any C source files. When `GCC` invokes the linker, it automatically adds `crt0.o` and many other libraries in the proper place. For example, the command `gcc exp.o` generates the following `ld` command:

```
<target-alias>$ ld –dc –dp –e start –X o –o a.out /usr/<target-alias>/lib/crt0.o –lc
```

In this command, the run-time initialization module `/usr/<target-alias>/lib/crt0.o` appears explicitly, in addition to requests to resolve references to the C library (the general runtime library). You can see what `ld` command is generating when you compile a program on your system, by invoking `GCC` with the `–v` (verbose) option.

9.9.2 Linker Options

The `GCC` passes any options it does not recognize to the linker. The most important options can therefore be placed directly on the `GCC` command line. These options are:

-o name.x

Instead of naming the executable output file `a.out or a.exe` (native builds under Cygwin), it names it `name.x`. Other popular naming extensions for embedded applications include: `name.elf`, `name.coff`, `name.srec`, and `name.bin`. Microcross has standardized all of its cross-executable names to the `.x` extension.

-lname

Link the program to the library named `libname.a`. The linker looks in the directories `/lib` and `/usr/lib` to find this library. Note that the GNU linker truncates the name's prefix and extension (i.e., `lib` and `.a` are not necessary for `libname.a` as a linked library file. If you create your own libraries to link into your program, you will need to name them `libname.a` and link in as `-lname`.

-Ldir

To find any libraries, look in the directory `dir` before looking in the standard library directories `/lib` and `/usr/lib`.

-s

Remove the symbol table from the executable output file. This makes the output file significantly smaller, but makes debugging nearly impossible. Therefore, this option should not be used until the program works successfully. Note that using the program `strip` has the same effect.

-x

Remove all local symbols from the output file. Global symbols (subprograms and common block names) remain in the output file. This reduces the object file's size. Ignored unless `–s` is specified.

-n

Make the text segment read-only.

-r

Create an object file that can be included in further linking runs (i.e., further passes through ld). Among other things, this inhibits undefined symbol messages (because the symbols may be defined in a later ld pass) and prevents ld from creating common storage immediately. If you wish to create common storage at this time, use the –d option also.

-e name

Use the symbol name as the entry point to the executable program. By default, the entry point is the beginning of the first object module. The GCC automatically links your object files with a run-time initialization module (/usr/lib/crt0.o) that starts your program and provides its initial entry point. If you run the linker separately, you must either put /usr/lib/crt0.o at the start of your object files, or provide your own entry point.

-M

Produce a load map that shows where each function is located in the resulting object file, where each section begins and ends, and what the value of each global symbol is. This option is usually used with GCC as follows: -Wl,-M=<mapfile-name>, where <mapfile-name> can be any name you specify.

-b format

Read object modules in the given format. To get a list of formats that ld understands, give the command objdump –I and near the bottom of the options list are the supported targets and architectures. This can be helpful in some cross-development situations. The –b option applies to all object files and libraries following it on the command line, until another –b option appears. In theory, you can use this feature to link objects from several different formats into a single executable.

-oformat format

Create object modules in the given format. Again, objdump –I gives you a list of formats that ld understands. The ld is configured to produce the most reasonable output format for its target machine. Its assumptions about what is “reasonable” are probably true about 99.99 percent of the time. But there may be special-purpose situations in which you would want another output format.

Here is an example of a customized ld command:

```
<target-alias>$ ld –r –o bigofile.o prog1.o prog2.o –lmylib
```

This command links the files prog1.o and prog2.o and the library file /usr/lib/libmylib.a. The resulting file is named bigofile.o; it can be linked further and may still contain unresolved references.

9.9.3 Linker Scripts

One advanced feature of the GNU linker is its ability to work from scripts written in its own command language. If you are a true masochist, you might be able to avoid running GCC altogether; you might be able to implement your own compiler as a linker script. There are a few situations in which you would actually need a linker script, but you should be aware that they exist for purposes like:

- Gaining tight control over the format of the output file – perhaps so an embedded application will fit into the smallest possible ROM, perhaps to optimize link order.
- Supporting an object format that ld does not provide – perhaps an object format of your own design, or an object format for some special-purpose operating system.

Microcross recommends starting with the default linker script and modifying it to meet your needs. To extract the standard linker script into a file, type the command:

```
<target-alias>$ ld –verbose >linker.ld
```

This command will create a file named `linker.ld,` and you will need to open this file in an editor to delete the header lines before and including`======` and the footer single line `======.` Then the script file can be saved or copied into a source directory and be used with the `GCC` option `-Wl,-T,linker.ld` to replace the standard linker script. Now, you can modify the linker script with confidence that you have a working startup script.

9.9.4 Link-Order Optimization

If you have done a lot of development work, you have probably noticed that the order in which you link your files can have a significant effect on performance. By changing the link order, you are changing the way the executable file lies in the instruction cache. The cache is a fast area of memory that stores pages of instructions so that the processor does not have to go back to slower parts of memory (or even worse, the disk) for every new instruction. Certain link orders minimize instruction-cache miss. The effect usually is not large, but in pathological cases (a really bad link order on a machine that is very sensitive to cache miss), link optimization can speed up runtime by 50 percent.

Unfortunately, not much can be said about link-order optimization. There are few rules, if any; and all the rules have many exceptions. In general, it is a good idea to place modules that call each other near each other on the command line. The reasoning behind the heuristic is simple: if function A makes many calls to function B, and both A and B can fit into the cache simultaneously, you will not pay a penalty for cache miss. Your best chance of fitting both functions into the cache simultaneously occurs when they are located next to each other in the object file. The `–M` option, which produces a load map, shows you how the object file is arranged; it will help you investigate cache performance.

Normally, rearranging the order of the object modules on the command line is sufficient for experimenting with link-order optimization. However, you can get very fine control over your executable file by writing a linker script. If you have a thorough knowledge of your target machine's architecture, you may be able to use this to your advantage – though you will probably reach the point of diminishing returns fairly quickly.

9.9.5 The C Runtime (crt0)

To link and run C or C++ programs, you need to define a small module, usually written in assembly as `crt0.s`, but sometimes written as a C file as `crt0.c`, to initialize hardware using C conventions before calling main. There are some examples available in the sources of Newlib C library; perform a simple search through the source tree for `crt0.s` code, as well as examples of system calls with sub-routines.

To write your own `crt0.s` or `crt0.c` module, you need specific information about your target – see `MC-Embedded-Systems.pdf` in the GNU X-Tools CD docs directory (also in Cygwin/docs) for details on how to create the C runtime environment.

9.10 Object Translation (ELF to Binary, SREC, etc.)

Most of the GNU X-Tools toolsuites produce an Extended Linker Format (ELF) object file as the default format, and it can be used during debug and testing; however, the final code needs to be stripped and translated to another format. It is beyond the scope of the user guide to explain all of the various formats, which include COFF, ECOFF, binary, srec, tekhex, ihex, symbolsrec, etc..

Here is a quick example of how to use objcopy; refer to the binutils documentation in the docs directory on the distribution CD for details on how to use. From the Shell, issue the following commands (substitute your target alias and bfdname as necessary).

e.g. (notional only):

```
$ xtools arm-linux
arm-linux$ cd /home/test
arm-linux$ gcc –g –o div.elf div.c
arm-linux$ objcopy –S –O binary div.elf div.bin
```

Note: The `-S` strips all symbols (including the debugging symbols put in the file by using the `-g` option) out to make the object as small as possible. The `-O binary` specifies the output target format. The input file is `div.elf` and the output file is `div.bin`.

9.11 Creating/Updating Libraries

The command `ar` creates libraries (or archives) of object modules. They are similar to the Unix / Linux utilities with the same names, except that you do not need a separate `ranlib`. This section gives a brief description of how to use these utilities in the GNU X-Tools command shell.

In naming a library for use with GNU X-Tools, always use the prefix `lib` and suffix extension `a` with your static library name; this is the GNU convention and is necessary for `GCC` to automatically scan the library at link time, so the syntax for linking a particular library, say `libm.a` the math library, is `-lm`.

To create a new library, use the `ar` command, as follows:

```
<target-alias>$ ar -rs lib<name>.a list-of-files
```

The option `r` indicates that the command `ar` should add the files named in the `list-of-files` to the library named `name`, creating a new library if necessary. If a file is mentioned twice in the `list-of-files`, `ar` includes it in the archive twice. The `s` option tells `ar` to produce an index for the archive; this is the function that `ranlib` would perform. If you include the `s` option whenever you create or modify a library, you will not need to use `ranlib`.

To update a library, use the command:

```
<target-alias>$ ar -rus lib<name>.a list-of-files
```

This compares the dates of any listed files with the version of the file in the library. If the file in `list-of-files` is more recent than the version contained in the library, `ar` substitutes the newer version for the older version. The `s` option updates the library's index.

To delete one or more files from a library, use the command:

```
<target-alias>$ ar -ds lib<name>.a list-of-files
```

This option deletes all the files found in `list-of-files`.

To extract one or more files from a library, use the command:

```
<target-alias>$ ar -x lib<name>.a list-of-files
```

This does not modify the library file itself. It extracts the files named in the `list-of-files` from the library, regenerating them in the current directory with their original names. Normally, the timestamp of the extracted files is the time at which `ar` recreated them. If you use the option `xo` instead of `x`, `ar` sets the timestamp of the extracted files to the time recorded in the archive.

You can still create an ordered (index-less) library with `ar` and invoke `ranlib` as a separate step if you want, which is the convention of older Unix Systems. However, there is no longer any good reason for doing that when using GNU tools.

9.12 GNU Libraries

If you are familiar with Unix / Linux and C programming, the libraries you will find in the GNU X-Tools development environment should not confuse you. The libraries you expect will all be there: standard I/O, the math library, the strings library, etc.. The libraries are ANSI C and POSIX compliant. Moreover, there are many functions that Unix / Linux programmers expect, but are not specified by either of these standards.

Some C functions have been standardized and found in the GNU environment, but system calls and math libraries are not the same as those found in a typical DOS / Windows environment. You will have to re-learn these functions. A good resource for learning the GNU Libraries is on the GNU X-Tools CD `Docs` directory: `MC-Libraries.pdf` and `Math_lib.pdf`.

Appendix 1. GNU X-Tools ARM Linux Toolsuite

MODEL (target alias):	ARM -- Advanced RISC Machines (alias: arm-linux)
MFR:	Multiple Manufacturers: ARM, Atmel, Freescale, Intel, NetSilicon, Philips, Samsung, Sharp, and many others
CPU TYPE:	RISC Von Neumann, 32-Bit Fixed Instructions, 16-Bit Thumb mode
GP REGS/SIZE:	16 / 32-Bit
ADDR SPACE:	64MB Linear -- Endian: Selectable
CYCLE RATE:	Core Dependent
APPROX INSTRUCTION THROUGHPUT (PIPELINED):	Depends on core variant
# INSTRUCTIONS:	11 TYPES
# ADDR MODES:	5 Load/Store
FP INSTRUCTIONS:	None
INTERRUPTS:	8 Exceptions, 2 External Interrupts (normal, fast)
VARIANTS:	ARM 2/250/3/6/60/600/610/620
	ARM 7/7m/7d/7dm/7di/7dmi/70/700/700i/710/71xxc/75xxfe/7tdmi
	ARM 8/810/9/9e/920/920t,,940t,9tdmi,
	StrongARM 110/1100
	Architectures: armv2, armv2a, armv3, armv3m, armv4, armv4t
	armv5, armv5t, armv5te, armv6j
ATTRIBUTES:	Based on current independent studies, the ARM is the most popular RISC processor for embedded applications. Its architectural specification is highly scalable, and a number of variants have been defined and produced by several manufacturers. As a result, a wide range of configurations and performance levels exist, giving embedded designers many flexible design options. The simple instruction set architecture is easy to learn, yet powerful and optimized for compilers.
REFERENCES:	www.arm.com

The ARM Toolsuite consists of the following tools:

Tool Name	Tool Description
arm-linux-gcc	GNU Compiler Collection (GCC)
arm-linux-g++	C++ compiler
arm-linux-as	GNU assembler (as)
arm-linux-ld	GNU linker (ld)
arm-linux-addr2line	Converts addresses to file names & line #
arm-linux-ar	Creates object code archives
arm-linux-nm	Lists symbols from object files
arm-linux-objcopy	Copies and translates object files
arm-linux-objdump	Displays information from object files
arm-linux-ranlib	Generates index to archive contents
arm-linux-readelf	Displays information about ELF objects
arm-linux-size	Lists file section sizes and total sizes
arm-linux-strings	Lists printable strings from files
Arm-linux-strip	Strips debug symbols from binaries
arm-linux-gdb	GDB debugger
arm-linux-gdbtk	Visual GDB

TARGET CPU DEPENDENT INFORMATION

This toolsuite comprises a complete development environment for the ARM processor family. Included in this section of the Appendix are the required compiler and linker flags, a list of debugger targets for both command line GDB and Visual GDB, machine compiler and assembler dependent options, and notes.

Required Compiler / Linker Flags

The following compiler/linker options are required to build for execution with the simulator.

```
CFLAGS = (none required)
LFLAGS = (none required)
```

Compiler Machine Dependent Options

Section 3 describes all of the common compiler, linker, and assembler options used in Microcross GNU X-Tools Linux edition; in addition, these -m options are defined for ARM architectures:

Option	Description
-mapcs-frame	Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying -fomit-frame-pointer with this option will cause the stack frames not to be generated for leaf functions. The default is -mno-apcs-frame.
-mapcs	This is a synonym for -mapcs-frame.
-mapcs-32	Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option. This option replaces the -m6 option of previous releases of the compiler and is now the default.
-mapcs-stack-check	Generate code to check the amount of stack space available upon entry to every function (that actually uses some stack space). If there is insufficient space available then either the function __rt_stkovf_split_small or __rt_stkovf_split_big will be called, depending upon the amount of stack space required. The run time system is required to provide these functions. The default is -mno-apcs-stack-check, since this produces smaller code. Must be used with -mapcs-frame.
-mno-sched-prolog	Prevent the reordering of instructions in the function prolog, or the merging of those instructions with the instructions in the function's body. This means that all functions will start with a recognizable set

Option	Description
	of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is `-msched-prolog`.
`-msoft-float`	This is the default, and it is not necessary to specify it. The Instruction Set Simulator (ISS) will only run code generated with the default software floating point emulation. Software floating point is the default.
`-mlittle-endian`	Generate code for a processor running in little-endian mode. This is the default for all standard configurations. Big endian is not supported with this toolsuite.
`-malignment-traps`	Generate code that will not trap if the MMU has alignment traps enabled. On ARM architectures prior to ARMv4, there were no instructions to access half-word objects stored in memory. However, when reading from memory a feature of the ARM architecture allows a word load to be used, even if the address is unaligned, and the processor core will rotate the data as it is being loaded. This option tells the compiler that such misaligned accesses will cause a MMU trap and that it should instead synthesize the access as a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary. This option is ignored when compiling for ARM architecture 4 or later, since these processors have instructions to directly access half-word objects in memory.
`-mno-alignment-traps`	Generate code that assumes that the MMU will not trap unaligned accesses. This produces better code when the target instruction set does not have half-word memory operations (implementations prior to ARMv4). Note that you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. The default setting for most targets is `-mno-alignment-traps`, since this produces better code when there are no half-word memory instructions available.
`-mcpu=<name>`	This specifies the name of the target ARM Linux processor. GCC uses this name to determine what kind of instructions it can use when generating assembly code. Permissible names are: `arm9`, `arm920`, `arm920t`, `arm9tdmi`, `arm9e`, `arm926ej-s`, `arm1020t`, `arm1136j-s`, `arm1136jf-s`, `mpcore`, `mpcorenovfp`, `arm1176jz-s`, `arm1176jzf-s`, `xscale`, `ep9312`.
`-mtune=<name>`	This option is very similar to the `-mcpu=` option, except that instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the CPU specified by a `-mcpu=` option. For some arm implementations better performance can be obtained by using this option.
`-march=<name>`	This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can use when generating assembly code. This option can be used in conjunction with or instead of the `-mcpu=` option. Permissible names are: `armv2`, `armv2a`, `armv3`, `armv3m`, `armv4` (the default setting), `armv4t`, `armv5`, `armv5te`, `armv6`, `armv6j`.
`-mfpe=<number>` `-mfp=<number>`	This specifies the version of the floating-point emulation available on the target. Permissible values are 2 and 3. `-mfp=` is a synonym for `-mfpe=` to support older versions of GCC.
`-mstructure-size-boundary=<n>`	The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissible values are 8 and

Option	Description
	32. The default value varies for different toolsuites. For the COFF targeted toolsuite the default value is 8. Specifying the larger number can produce faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with the other value, if they exchange information using structures or unions. Programmers are encouraged to use the 32 value as future versions of the toolsuite may default to this value.
`-mabort-on-noreturn`	Generate a call to the function abort at the end of a noreturn function. It will be executed if the function tries to return.
`-mlong-calls` `-mno-long-calls`	Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction. Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the `short-call` attribute, functions that are inside the scope of a `#pragma no_long_calls` directive and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the long-call attribute or the section attribute, and functions that are within the scope of a `#pragma long_calls` directive, will always be turned into long calls. This feature is not enabled by default. Specifying `--no-long-calls` will restore the default behavior, as will placing the function calls within the scope of a `#pragma long_calls_off` directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.
`-msingle-pic-base`	Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The run-time system is responsible for initializing this register with an appropriate value before execution begins.
`-mpic-register=<reg>`	Specify the register to be used for PIC addressing. The default is r10 unless stack checking is enabled, when r9 is used. Must be used with –fpic option to be valid.

`-mpoke-function-name`	Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this:
	t0
	.ascii "arm_poke_function_name", 0
	.align
	t1
	.word 0xff000000 + (t1 - t0)
	arm_poke_function_name
	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	When performing a stack backtrace, code can inspect the value of pc stored at fp + 0. If the trace function then looks at location pc - 12 and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length ((pc[-3]) & 0xff000000).

Compiler-Defined Symbols and Standard Search Directories

From the command line in GNU X-Tools Shell, type:

```
$ xtools arm-linux
arm-linux$ gcc -v -E -
```

You will get all of the compiler-defined symbols printed out on the screen. Press `Ctrl-C` to exit this mode.

Assembler Machine Dependent Options

The following options are available when as is configured for the ARM processor family.

Options	Description
-marm7[tdmi] -marm8[10] -marm9,920,9tdmi -mstrongarm110[0]] -marmv4 | -marmv4t -marmv5 | -marmv5te -marmv6 | -marmv6j	Arm 7 processors Arm 8 processors Arm 9 processors StrongARM processors Arm architectures
-mfpe-old	No float load/store multiples
-mfpa10 | -mfpa11	FPA10 and 11 co-processor
-mapcs-32	Select which procedure calling convention is in use: 32-bit APCS
-EL -EB	Select little-endian (-EL) output – the default Select big-endian (-EB) output

GNU ARM Assembler Quick Reference

A summary of useful commands and expressions for the ARM architecture using the GNU assembler is presented briefly in the concluding portion of this Appendix. Each assembly line has the following format:

```
[<label>:]  [<instruction or directive>}  @ comment
```

Unlike the ARM assembler, using the GNU assembler does not require you to indent instructions and directives. Labels are recognized by the following colon instead of their position at the start of a line. An example follows showing a simple assembly program defining a function `add` that returns the sum of two input arguments:

```
.section .text, "x"

.global     add                 @ give the symbol add external linkage

add:
      ADD   r0, r0, r1          @ add input arguments
      MOV   pc, lr              @ return from subroutine

                                @ end of program
```

GNU Assembler Directives for ARM

The follow is an alphabetical listing of the more command GNU assembler directives.

GNU Assembler Directive	Description
.ascii "<string>"	Inserts the string as data into the assembly (like DCB in armasm).
.asciz "<string>"	Like .ascii, but follows the string with a zero byte.
.balign <power_of_2>	Aligns the address to <power_of_2> bytes. The assembler aligns by

GNU Assembler Directive	Description
{,<fill_value> {,<max_padding>} }	adding bytes of value <fill_value> or a suitable default. The alignment will not occur if more than <max_padding> fill bytes are required (similar to ALIGN in armasm).
.byte <byte1> {,<byte2>} …	Inserts a list of byte values as data into the assembly (like DCB in armasm).
.code <number_of_bits>	Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly (similar to CODE16 and CODE32 in armasm).
.else	Use with .if and .endif (similar to ELSE in armasm).
.end	Marks the end of the assembly file (usually omitted).
.endif	Ends a conditional compilation code block – see .if, .ifdef, .ifndef (similar to ENDIF in armasm).
.endm	Ends a macro definition – see .macro (similar to MEND in armasm).
.endr	Ends a repeat loop – see .rept and .irp (similar to WEND in armasm).
.equ <symbol name>, <value>	This directive sets the value of a symbol (similar to EQU in armasm)
.err	Causes assembly to halt with an error.
.exitm	Exit a macro partway through – see .macro (similar to MEXIT in armasm)
.global <symbol>	This directive gives the symbol external linkage (similar to EXPORT in armasm).
.hword <short1> {,<short2>} …	Inserts a list of 16-bit values as data into the assembly (similar to DCW in armasm).
.if <logical_expression>	Makes a block of code conditional. End the block using .endif (similar to IF in armasm). See also .else.
.ifdef <symbol>	Include a block of code if <symbol> is defined. End the block with .endif.
.ifndef <symbol>	Include a block of code if <symbol> is not defined. End the block with .endif.
.include "<filename>"	Includes the indicated source file (similar to INCLUDE in armasm or #include in C).
.irp <param> {,<val_1>} {,<val_2>} …	Repeats a block of code, once for each value in the value list. Mark the end of the block using a .endr directive. In the repeated code block, use \<param> to substitute the associated value in the value list.

.macro <name> {<arg_1}
{,<arg_2>} … {,<arg_N>}
	Defines an assembler macro called <name> with N parameters. The macro definition must end with .endm. To escape from the macro at an earlier point, use .exitm. These directives are similar to MACRO, MEND, and MEXIT in armasm. You must precede the dummy macro parameters by \. For example:
	
	.macro SHIFTLEFT a, b
	  .if \b < 0
	    MOV \a, \a, ASR #-\b
	    .exitm
	  .endif
	  MOV \a, \a, LSL #\b
	.endm

.rept <number_of_times>	Repeats a block of code the given number of times. End with .endr.
<register_name> .req <register_name>	This directive names a register. It is similar to the RN directive in armasm except that you must supply a name rather than a number on the right (e.g., acc .req r0).
.section <section_name> {,"<flags>"}	Starts a new code or data section. Sections in GNU are called .text, a code section, .data, an initialized data section, and .bss, an uninitialized data section. These sections have default flags, and the linker understands the default names (similar directive to the armasm directive AREA). The following are allowable .section flags for ELF

GNU Assembler Directive	Description
	format files: <Flag> Meaning a allowable section w writable section x executable section
`.set <variable_name>, <variable_value>`	This directive sets the value of a variable. It is similar to SETA in armasm.
`.space <number_of_bytes> {,<fill_byte>}`	Reserves the given number of bytes. The bytes are filled with zero or <fill_byte> if specified (similar to SPACE in armasm).
`.word <word1> {,<word2>} …`	Inserts a list of 32-bit word values as data into the assembly (similar to DCD in armasm).

Assembler Special Characters / Syntax

Inline comment char: `@`
Line comment char: `#`
Statement separator: `;`
Immediate operand prefix: `#` or `$`

Register Names

```
General registers:          %r0 - %r15        ($0 = const 0)
FP registers:               %f0 - %f7
Non-saved (temp) regs:      %r0 - %r3, %r12
Saved registers:            %r4 - %r10
Stack ptr register:         %sp
Frame ptr register:         %fp
Link (retn) register:       %lr
Program counter:            %ip
Status register:            $psw
Status register flags:      xPSR
(x = C current)             xPSR_all
(x = S saved )              xPSR_f
                            xPSR_x
                            xPSR_ctl
                            xPSR_fs
                            xPSR_fx
                            xPSR_fc
                            xPSR_cs
                            xPSR_cf
                            xPSR_cx
                            .. and so on
```

Arm Procedure Call Standard (APCS) Conventions

```
Argument registers:     %a0 - %a4     (aliased to %r0 - %r4)
Returned value regs:    %v1 - %v6     (aliased to %r4 - %r9)
```

Addressing Modes

`rn` in the following refers to any of the numbered registers, but not the control registers.

`addr` Absolute addressing mode

`%rn`	Register direct
`[%rn]`	Register indirect or indexed
`[%rn,#n]`	Register based with offset
`#imm`	Immediate data

Machine Dependent Directives

`.arm`	Assemble using arm mode
`.thumb`	Assemble using thumb mode
`.code16`	Assemble using thumb mode
`.code32`	Assemble using arm mode
`.force_thumb`	Force thumb mode (even if not supported)
`.thumb_func`	Mark entry point as thumb coded (force bx entry)
`.ltorg`	Start a new literal pool

Opcodes

For detailed information on the machine instruction set, see this manual:

ARM Architecture Reference Manual, Addison-Wesley ISBN 0-201-73719-1

Here is a recommended book to get a lot of system developer information on the ARM architecture.

ARM System Developer's Guide, Morgan Kaufmann Publishers ISBN 1-55860-874-5 (alk.paper), authors: Andrew N. Sloss, Dominic Symes, Chris Wright, 2004

NOTES & ERRATA

Appendix 2. Common Minicom Problems

If you are having problems with Minicom, first try clearing all settings for Modem and dialing under Configure Minicom. Save the settings; this requires root permissions.

Problem	Solution
Error message: *Device /dev/ttyS0 access failed: Permission denied.*	Permissions for device are set improperly. As root user, enter # chmod o+rw /dev/ttyS0
No terminal activity.	• In Configure Minicom --> Serial port setup, check that Serial Device is consistent with the port you are using. • In Configure Minicom --> Serial port setup, ensure that Hardware Flow Control and Software Flow Control are both set to No. • If possible, connect the serial cable to another serial port on a Linux workstation and verify data can be exchanged over the serial cable in both directions.
Settings revert to original when Minicom is restarted.	Make sure you use Configure Minicom --> Save setup as dfl after making changes. To do this, you must run Minicom as the root user.

Appendix 3. Glossary

TERM	Description
ARM Ltd	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	Lucent Technologies, Inc.'s code used to load code to flash and set boot options. See uMON documentation.
BOOTP	Network protocol allowing a network device to obtain an IP address and other information from a server.
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.
CLUT	Color look-up table
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 hardware.
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."
ELF	ELF (Executable and Linking Format) is a binary format originally developed by USL (UNIX System Laboratories). Because of its increased flexibility over the older a.out format that Linux previously used, the GCC and C library developers decided to move to using ELF as the Linux standard binary format also.
File system	The directory structure (beginning with "/") comprising configuration files, applications, libraries, and an operating system.
GDB	Gnu DeBugger. Allows you to see what is going on inside another program while it executes -- or what another program was doing at the moment it crashed.
GIO or GPIO	General Purpose Input / Output hardware lines.
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 Module 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."
GUI	Graphical user interface.
Host Computer	The Linux-based desktop system used for software development.
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

TERM	Description
	prompt shell is "$" for normal users and "#" for root.
IP	Internet Protocol.
JFFS	The Journaling Flash File System, developed by Axis Communications in Sweden, is aimed at providing a crash/power-down-safe file system for diskless 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.
Linux Workstation	The computer running the Linux operating system that is used for developing, compiling, and building the kernel and target file system that is loaded and executed on the SBC.
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 MS-DOS Telix program.
MMU	The Memory Management Unit (MMU) is a component of a processor architecture. The MMU is responsible for mapping physical addresses defined by the hardware configuration into the virtual addresses the are used by the Linux-based software. All Linux kernels require that the processor support an MMU; uClinux is a special version of Linux for processors that do not support an MMU.
NFS	The Network File System (NFS) allows machines to mount a disk partition on a remote machine as if it were on a local hard drive. This allows for fast, seamless sharing of files across a network.
OSD	On-Screen Display.
SBC	Single Board Computer.
SDK	Software Development Kit.
SOC	System-On-a-Chip.
Target File System	The file system for a particular target hardware.
Target Hardware	The hardware running Linux.
TFS	Trivial File System. This is the small flash file system managed by the uMON bootloader.
TI	Texas Instruments, Inc.
Toolchain	Tools and utilities used to build applications and other code for the target hardware.
UI	User Interface.
uMON	MicroMonitor. The Lucent Technologies, Inc. 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, uMON will typically transfer control to the stored system.
x86	Any processor compatible with the Intel 386, 486, or Pentium family.
XIP	Execute In Place.

Appendix 4. Additional Resources

ARM, Ltd
www.arm.com

Atmel Corporation
www.atmel.com

BusyBox, the Swiss Army Knife for Embedded Linux
www.busybox.net

Cogent Computer Systems, Incorporated
www.cogcomp.com

Freescale Semiconductor, Inc.
www.freescale.com

Linux documentation, including guides and FAQs
www.linuxdoc.org

Linux newbie.org for new Linux users
www.linuxnewbie.org

Linux kernel organization that controls new releases and has archives of old ones
www.kernel.org

Memory Technology Device (MTD) for details on memory (Flash) devices
www.linux-mtd.infradead.org

Microcross, Incorporated
www.microcross.com