SnapGear Linux for LEON

Manual: SnapGear Linux for LEON

Written by Daniel Hellström

LINUX-SNAPGEAR
Version 1.36.0
December 2007

Första Långgatan 19
413 27 Göteborg
Sweden

tel: +46 31 7758650
fax: +46 31 421407
www.gaisler.com

Table of Contents

1 INTRODUCTION

LINUX support for LEON2 and LEON3 is provided through a special version of the SnapGear Embedded Linux distribution. SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. The LEON port of SnapGear supports both MMU and non-MMU LEON configurations, as well as the optional V8 mul/div instructions and floating-point unit (FPU). A single cross-compilation tool-chain is provided which is capable of compiling the kernel and applications for any configuration.

LEON Linux has support for symmetric multi-processing (SMP), it has not been extensively tested yet, but work is in progress.

1.1 LEON Linux

There are two different versions of the Linux kernel in the Gaisler SnapGear distribution, namely 2.6.x and 2.0.x. The 2.0 version is modified for use with MMU-less LEON systems, it is called micro controller Linux – µCLinux. 2.6.x has support for MMU systems only, the kernel is from kernel.org with LEON specific patches and additional drivers mainly for GRLib hardware.

The Linux kernel can be configured using a graphical interface. One can remove drivers and features to save space. On LEON3 systems the AMBA plug&play information is used to detect devices and load their respective software drivers. LEON2 uses hard coded addresses to find its devices. During configuration the processor type is selected, LEON2 or LEON3, it is done from the Linux kernel configuration GUI and in the main SnapGear GUI.

Multi processor LEON3 systems are supported by Linux 2.6.21.1, symmetric multi-processing support (SMP) can be enabled through the Linux kernel configuration.

The Linux kernel can be used for other projects that need not be based on SnapGear. The boot loader will still be needed but it is possible to create custom projects with custom file systems. Even though it is out of the scope of this document, it is described how to set up Linux with other distributions via NFS.

1.2 SnapGear Linux

SnapGear Linux is a full source package, containing kernel, libraries and application code for rapid development of embedded Linux systems. It is configured from a graphical interface similar to the Linux 2.4 kernel configuration utility.

1.3 Boot loader for LEON Linux

A small boot loader has been incorporated into the SnapGear Linux software distribution, it is designed especially for the LEON processors, both SMP and uniprocessor systems. It is capable of passing arguments to any of the Linux kernels and initialize low level hardware. The main purpose of the boot loader should be to initialize basic hardware, such as memory controllers and console output for debugging, before launching LEON Linux.

The SnapGear graphical interface as been extended to allow users to set boot loader parameters in an easy fashion. The boot loader is stored in a non-volatile memory at the address where the LEON processor reads its first instructions to be executed, usually stored in flash at address 0.

During the development process *grmon* may be used to load the resulting image into RAM directly, thus bypassing the flash. This shortens the development time drastically. Using this method only the last part of the boot loader is executed, it is often referred to as “stage 2”. Instead, grmon initializes the hardware before running stage 2.

1.4 LEON simulator to speed up the development process

There are simulators available for LEON and most of its peripherals, TSIM and the multiprocessor simulator GRSIM. See www.gaisler.com for more information about simulators.

1.5 Obtaining the software

The Software is free of charge and distributed under the GPL licence. The software bundle can be downloaded from Gaisler's homepage: www.gaisler.com under the downloads section.

1.6 Supported hardware

Below is a list of supported hardware in addition to the standard kernel:

- LEON2, with or without MMU, FPU, MUL/DIV.
- LEON3, with or without MMU, FPU, MUL/DIV.
- LEON3 multi processor systems, SMP
- APBUART
- GPTIMER
- GRETH 10/100 and Gbit
- OpenCores 10/100 Ethernet MAC
- SMC91x 10/100 Ethernet MAC
- APBPS2
- APBVGA
- GRUSBHC
- GRVGA
- ATACTRL
- GRPCI
- GRETH over PCI
- GR/OpenCores I2CMST
- SPICTRL

Note that new hardware is being added constantly.

1.7 Support

For support, contact the Gaisler Research support team at support@gaisler.com

2 INSTALLING GNU TOOLCHAIN AND LEON LINUX

SnapGear has been split up into two different distributions, one for Linux 2.6 development and one for Linux 2.0 development.

The toolchain is a composition of several utilities used in the compilation process. It is intended to be used with Linux only, the most important utilities are the GNU GCC compiler and linker. The toolchain is a cross-compiler toolchain making it possible to compile LEON SPARC Linux binaries on an ordinary PC running Linux. The toolchains are distributed as a binary package freely available at www.gaisler.com.

2.1 Selecting Toolchain

For Linux 2.0 selecting toolchain is simple, as only one is available, the *sparc-linux-3.2.2*. For Linux 2.6 however, one select toolchain based on what C Library is going to be used, installing multiple toolchains cause no harm. The two toolchains available for Linux 2.6 are *sparc-linux-3.4.4* and *sparc-uclinux* as indicated by the table below. The next chapter gives a short introduction to the two different C Libraries.

Name	Description	Location
sparc-linux-3.2.2	Linux 2.0 and 2.6 GNU LibC toolchain	linux/linux-2.0/toolchains/sparc-linux-3.2.2
sparc-linux-3.4.4	Linux 2.6 GNU LibC toolchain	linux/linux-2.6/toolchains/sparc-linux-3.4.4
sparc-uclinux-3.4.4	Linux 2.6 µClibC toolchain	linux/linux-2.6/toolchains/sparc-uclinux-3.4.4

Table 2.1: Toolchain description

The locations described in table 2.1 are all relative to the Gaisler FTP site ftp://ftp.gaisler.com/gaisler.com.

2.2 Installing the toolchain

The installation process for the different toolchains is the same, it is only the names and paths that differ. All toolchains must be installed to /opt and the path to the toolchain binary directory (/opt/sparc-[uc]-linux-3.x.x/bin) added to the shell PATH variable. Below is an example of how to install the sparc-linux-3.2.2 toolchain.

Install by extracting the toolchain into /opt:

```
$ mkdir /opt
$ cd /opt
$ tar -jxf /path/to/toolchain/sparc-linux-1.0.0.tar.bz2
```

Add the toolchain to the PATH variable preferably in a shell start up script. For bash shells the following is added to ~/.profile:

```
export PATH=$PATH:/opt/sparc-linux/bin
```

After installing the toolchain it is possible to cross compile applications for SPARC LEON Linux:

args.c:

```
int main(int argc, char *argv[]){
        printf("%s: you passed %d argument(s)\n",argv[0],argc-1);
        return 0;
}
```

Compile args by running:

```
$ sparc-linux-gcc -o args args.c
```

or

```
$ sparc-uclinux-gcc -o args args.c
```

From the ELF header it can be read that the output binary is a SPARC binary:

```
$ file args
args: ELF 32-bit MSB executable, SPARC, version 1 (SYSV), dynamically
linked (uses shared libs), not stripped
```

Running the binary on a SPARC Linux host results in:

```
$ ./args Gaisler
./args: you passed 1 argument(s)
```

2.3 Installing SnapGear for LEON Linux 2.0

Install the SnapGear distribution by extracting it:

```
$ mkdir ~/linux
$ cd ~/linux
$ tar -xjf /path/to/dist/snapgear-2.0-p36.tar.bz2
$ ls
 snapgear-2.0-p36
```

2.4 Installing SnapGear for LEON Linux 2.6

Install the SnapGear distribution by extracting it:

```
$ mkdir ~/linux
$ cd ~/linux
$ tar -xjf /path/to/dist/snapgear-2.6-p36.tar.bz2
$ ls
 snapgear-2.6-p36
```

3 CONFIGURING LINUX

SnapGear comes with an easy to use graphical interface similar to the Linux kernel's configuration utility. From the GUI it is possible to select processor, Linux version, C library and what applications will be included into the root file system (ROMFS image) accessed by Linux during runtime. It is also possible to configure the boot loader parameters and configure the Linux kernel.

The GUI can be launched by doing a 'make xconfig'. The main configuration menu should appear:

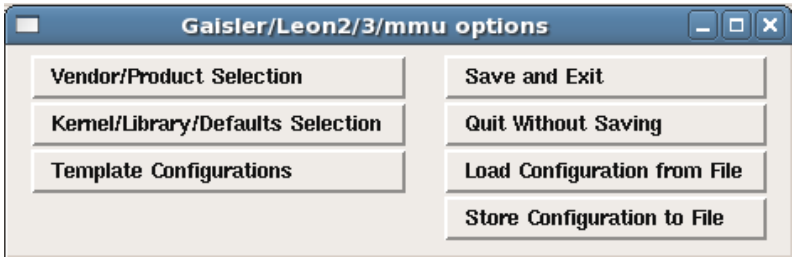

Illustration 3.1: SnapGear main configuration GUI

3.1 Processor type and MMU

Selecting processor is mandatory, the boot loader needs to know how to initialize low level hardware, which in some cases are processor dependant. It is also important to select MMU support if memory protection hardware is available for use.

MMU provides memory protection between kernel and user space, and also in between user space processes. With MMU a faulty process cannot affect another process memory in a destructive manner as may be the case without MMU.

Linux 2.6 cannot run without MMU, whereas µCLinux runs without MMU, indicated in the table below.

Version / MMU	MMU	MMU-less
2.0.x	N/A	µCLinux 2.0.x
2.6.x	Linux 2.6	N/A

Table 3.1: Linux and MMU

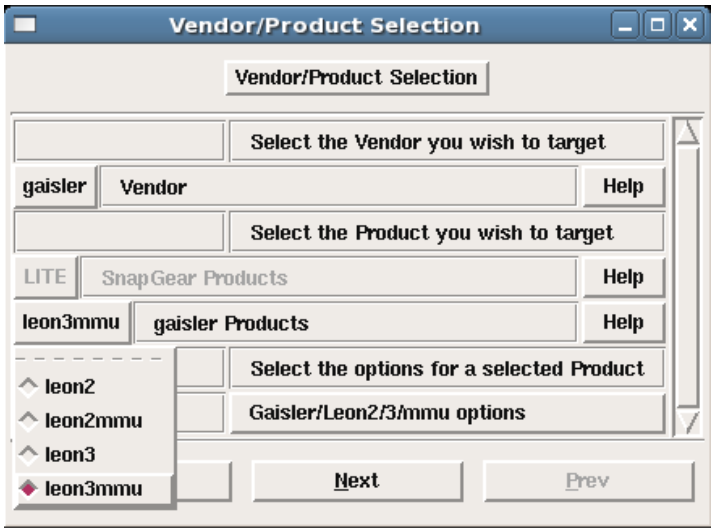

Illustration 3.2: Vendor/Product Selection

3.2 C library

Two different C libraries are available for selection from the graphical “xconfig” utility. The libraries differ in binary size and one of them, µClibC, supports MMU-less systems. Table 3.2 outlines which C library is available for which version of Linux. For configurations that supply their own root filesystem outside of SnapGear (such as via NFS), a 'none' option is also provided to entirely omit the compilation of a C system library.

Version / LibC	LibC (GNU)	µController LibC
µCLinux 2.0.x	N/A	Small footprint
Linux 2.6.x	Normal footprint	Small footprint

Table 3.2: LibC selection possibilities

3.2.1 Static vs Dynamic linking

When linking an application static, all code used from libraries are included into the output binary. This makes the binary bigger. Linking all binaries static will make the same code appear in multiple locations (in each binary that uses it) both in the file system and in main memory during execution.

Dynamic linking however, runs a dynamic linker during the start of the application on the target hardware, this results in smaller main memory usage as code is shared between applications. Since the same code may appear in different addresses in different applications, the code must be position independent. Generating position independent machine code makes the code a bit slower and slightly bigger, it is done by passing the argument *-fPIC* to the GNU compiler.

The more applications a file system has, the more space can be saved using dynamically linked binaries. Depending on needs and resources, different embedded systems choose different linking strategies.

3.2.2 Toolchains for Linux 2.6

The available SPARC LEON toolchains for Linux 2.6 both contains a precompiled C Library. The library has been built four times for CPUs with different combinations of FPU and mul/div hardware. Two different toolchains are available the sparc-linux toolchain includes the GNU C library whereas the sparc-uclinux toolchain has µClibC built in. The two toolchains are based on GCC 3.4.4. Installing both toolchains will not conflict. See previous chapter for installation instructions.

3.2.3 Toolchains for Linux 2.0

For Linux 2.0 there are one toolchain available, based on GCC 3.2.2. For Linux 2.0 SnapGear projects the toolchain's C Library isn't used, but the µClibC Library included in the SnapGear release. See installation instructions in previous chapter.

3.3 Kernel version

Linux kernel version can be selected from the Kernel/Library sub menu. As described earlier in the introduction there are two variants of the Linux kernel within the SnapGear LEON Linux distribution. One is intended to be used with MMU-less systems and the other is based on an older version of the Linux kernel, 2.0. For MMU based systems, Linux 2.6.x is available.

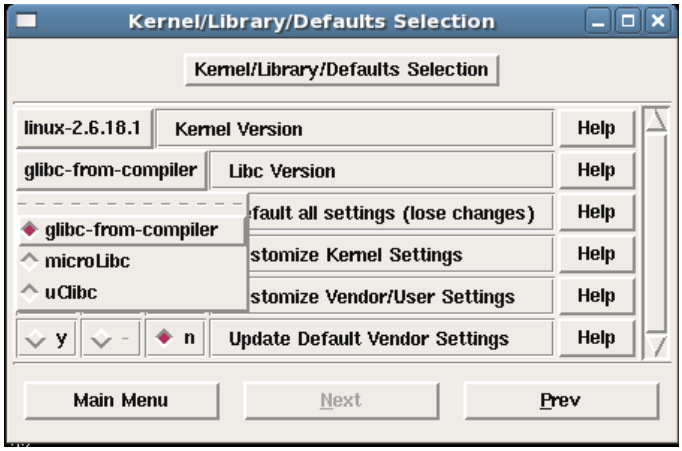

Illustration 3.3: Kernel, LibC and application selection menu

3.4 Configuring the boot loader

The boot loader sets up low level hardware before entering the Linux kernel. The parameters of the boot loader can be set using the main SnapGear GUI. Below is a description of available boot loader parameters.

Name	Description
SPARC v8, MUL/DIV	When “Yes” the compiler emits hardware integer multiplier instruction. If the LEON processor has been configured without hardware multiplier select “No”, the compiler will then generate SPARC v7 compatible code instead (without any MUL or DIV instructions).
FPU support	Select “Yes” if LEON has a Floating point unit (FPU).
Clock frequency	The frequency is needed for the boot loader to calculate timing values. Value is in kHz.
Baudrate	The baud rate of the first serial terminal. Typically 38400 baud. It is favourable to use the same baud rate for the Linux kernel.
UART Loopback on	Enables the loopback mode of the UART, every character sent will be recieved.
UART hardware flowctrl	Hardware flow control will be used.
In memory root file system	Romfs. Read only root file system. Created from snapgear-pxx/romfs. This option is valid for Linux 2.0.x.
	Initramfs. A read and write root file system put in main memory. See linux-2.6.x/Documentation/filesystems/ ramfs-rootfs-initramfs.txt for more information.
	Custom, an initramfs is created, similar to initramfs. The difference is that with this option one can make an file system externally and provide a description how it is created. The description is a text file as described in the Linux kernel documentation at linux-2.6.x/ Documentation/filesystems/ramfs-rootfs-initramfs.txt.
	None. When specified the linker doesn't include the any root file system. This means that Linux must read the root file system from an alternative location, for example NFS or ATA-disk.
Custom initramfs source	The text file used to create an custom filesystem, not necessarily based on SnapGear. See linux-2.6.x/ Documentation/filesystems/ramfs-rootfs-initramfs.txt for details.
Kernel command line	The kernel parameters is specified by a string. The string is interpreted by Linux during the boot sequence. See linux-2.6.x/Documentation/kernel-parameters.txt for details. Default: console=ttyS0,34800 Note: When using Linux 2.0.x the kernel command line is set from within the kernel configuration tool under "General setup".
ROM bank size	Bank size of flash
ROM rws	Number of ROM/Flash read wait states
ROM wws	Number of ROM/Flash write wait states
Enable write cycles to PROM	Boot loader makes it possible to write to FLASH without tampering with memory configuration registers.
RAM type	RAM type to be used. SRAM or SDRAM
Alternative physical kernel address	It is possible to manually select an address where the kernel will be started from. The base address of the stack can be changed as well. One can this way make room for a custom data area, the Linux kernel will only use the memory between the kernel base address and the stack base.

Table 3.3: Boot loader parameters

3.4.1 Symmetric multi-processing

Multi LEON3 processor systems is supported by the boot loader. The multi processor support is controlled from within the Linux 2.6.x kernel configuration GUI, once SMP is enabled the boot loader's SMP support is also included when built the next time.

See 2.6.x configuration section.

3.5 Configuring the 2.6.x kernel

From the graphical interface it is possible to configure the selected kernel by selecting “y” at “Customize Kernel Settings” and saving the new settings. After the main GUI has been closed and settings have been saved a second GUI will appear that is specific for the kernel selected. The kernel configuration GUI for Linux 2.6 is shown below. Details on how to configure the kernel can be found in the linux-2.6.x/Documentation directory. In later chapters some common configurations will be presented.

The following sections will be used to describe some of the Linux kernel's settings specific to LEON Linux.

Features and drivers often depend on other features, the dependencies is sometimes not trivial. The dependencies can be seen in a C programming similar syntax by enabling “Show debug info” from the Option menu. The dependency can be seen down to the right by selecting the feature in question.

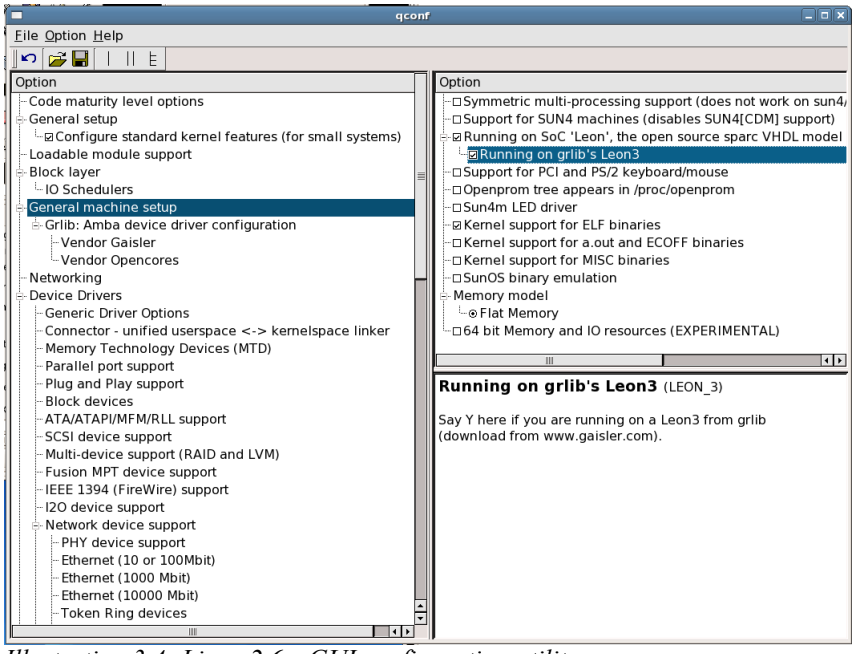

Illustration 3.4: Linux 2.6.x GUI configuration utility

3.5.1 LEON processor type

The configuration of the processor type is done separately for the Linux boot loader and the kernel itself. This is done to eliminate the kernel's dependency on SnapGear, LEON Linux can be used without SnapGear.

LEON must always be selected but LEON_3 is only selected for LEON3 targets. Selecting LEON_3 will result in binary unable to run on LEON2 processors and vice versa.

Certain hardware is only available for LEON3, their drivers will be invisible when LEON2 is selected.

3.5.2 Symmetric multi-processing support

LEON Linux has support for multi processor systems, SMP can be enabled for LEON3 systems under “General machine setup”. The LEON Linux boot loader is updated to initialize the SMP system correctly automatically when enabling SMP from the Linux kernel configuration GUI.

SMP is only available for LEON3 processors.

3.5.3 Gaisler AMBA Plug&Play procfs support

Procfs is a "pseudo file system" that is normally mounted onto */proc*. The procfs is directly linked to the kernel's internals and can display information relevant to the system's operation. Enabling AMBA_PROC makes a directory */proc/bus/amba* appear that can display information about devices on the AMBA bus.

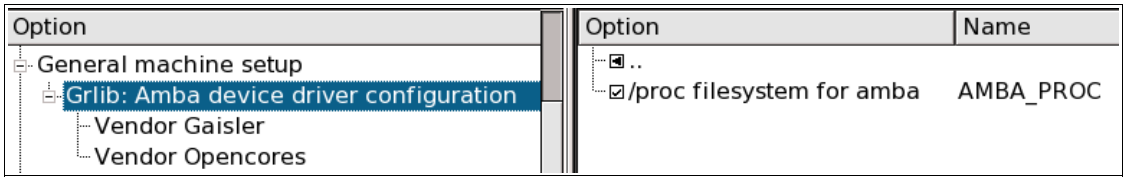

Illustration 3.5AMBA procfs support

3.5.4 GRLib APBUART (LEON3)

If a serial UART is to be used select GRLIB_GAISLER_APBUART, be sure to select GRLIB_GAISLER_APBUART_CONSOLE if one of the serial controllers are interfaced to a console. One of the serial terminals can be set up as the system console via the kernel boot parameter, ex: console=ttyS0,38400 selects the first serial channel to act as the system console. The serial terminals will be available under /dev/ttySx.

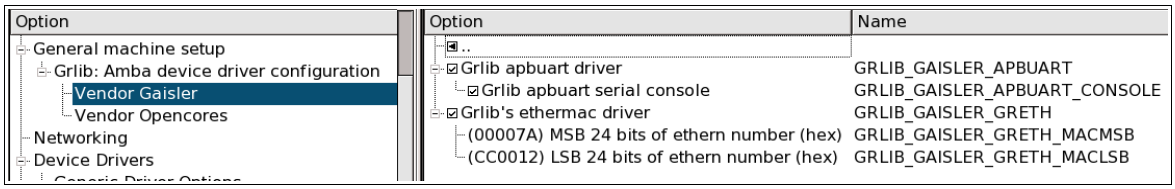

Illustration 3.6: APBUART and GRETH Linux configuration.

3.5.5 LEON Serial (LEON2)

For LEON2 systems the serial driver can be found under "Character devices / Serial drivers". Selecting SERIAL_LEON enables the serial driver, the serial devices can be used to communicate over the serial line

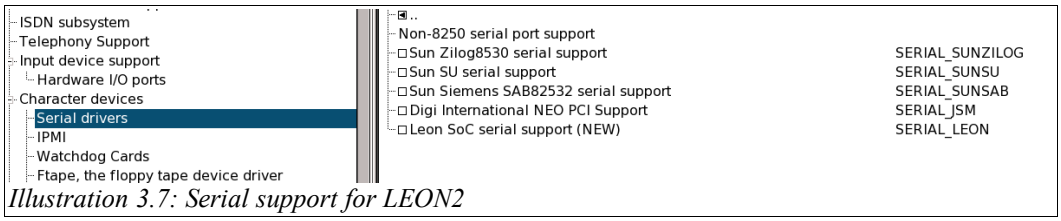

Illustration 3.7: Serial support for LEON2

with an arbitrary protocol. The serial lines can be accessed from /dev/ttySx.

3.5.6 GRLib GRETH 10/100/1000

Select GRLIB_GAISLER_GRETH to enable the 10/100 Ethernet MAC or for grlib professional users, the 10/100/1000 Ethernet MAC.

The Ethernet MAC address of the GRETH MAC can also be edited directly from the GUI. The address is made out of an unique 6 byte sequence. One can edit the 24 most significant bits (MSB) and the 24 least significant bits (LSB) from GRLIB_GAISLER_GRETH_MACXSB.

"Networking", "Network devices" and "TCP/IP" must be selected for standard network communication to be available.

3.5.7 GRLib OpenCores Ethernet MAC

In the same way as the GRETH is configured the OpenCores MAC can be selected and it's MAC address can be edited. The OpenCores MAC is found under "General machine setup / .. / Vendor OpenCores".

3.5.8 SMC 91x Ethernet MAC

The SMC91x driver can be found under “Network device support / Ethernet 10/100”. The Ethernet MAC address must be configured to an unique address as previous MAC controllers.

The chip is not a Plug&Play AMBA device and can therefore not be automatically detected by the driver. Both address and IRQ number must be given to the driver for it to operate correctly.

3.5.9 GRLib GRETH 10/100/1000 over PCI

Select GRETH_PCI from “Network device support / Ethernet 10/100” to include a driver for Gaisler Ethernet MAC connected via PCI. Usually GRETH is connected via the AMBA interface and is included as previously described.

PCI support must also be enabled, see the sub section GRLib PCI support below.

3.5.10 GRLib OpenCores I²C-master

Enable “I2C support” and “I2C support / I2C Hardware Bus support / OpenCores I2C Controller”. The driver will use the GRLib wrapper's interface when the kernel is configured to run on a Leon SoC. To use the I²C bus, selections should also be made under “I2C support / I2C Algorithms” and “I2C support / Miscellaneous I2C Chip support”, depending on which peripherals that are present in the system.

3.5.11 GRLib PCI support

GRPCI is a bridge between the AMBA bus and the PCI bus. The GRPCI core is mainly used to connect off chip controllers to the LEON system. It can be enabled by selecting “PCI support” under “General machine setup”.

Once the GRPCI driver detects the GRPCI core by probing the AMBA Plug&Play bus, it initializes the core and starts scanning the PCI bus for additional controllers.

3.5.12 GRLib GRPS2

Keyboard and mouse drivers are available for the Gaisler PS/2 controller. One can enable GRPS2 under “input device / Hardware I/O ports”. Input devices that use the PS/2 driver such as keyboard and mouse drivers are enabled under “Input device support”, MOUSE_PS2 and INPUT_KEYBOARD.

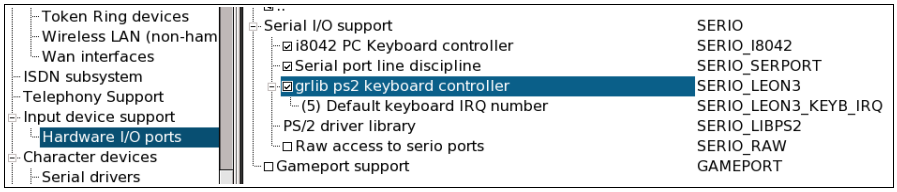

Illustration 3.8: Adding support for GRPS2 controller.

To separate keyboard and mouse PS/2 devices from each other the IRQ number of the keyboard must be specified. This is because the PS/2 controllers are identical to the kernel. PS/2 controllers found with an IRQ not matching the keyboard IRQ number are assumed to be a PS/2 controller connected to a mouse.

The default keyboard IRQ number may be overridden by the kernel command line option:

```
grps2=kbdirq:irqno  (irqno=0..15)
```

3.5.13 GRLib SPICTRL

The Gaisler SPI controller is enabled by selecting “SPI support” and “SPI support / Gaisler Research SPI Controller”. The driver automatically detects the number of available slave select lines.

Adding support for specific SPI devices requires editing of the SPI initialization code found in *linux-2.6.21.1/arch/sparc/kernel/leon_spic.c*. This is necessary since the available SPI devices are normally hard coded in platform specific code, and GRLib is used in a wide range of systems.

The current initialization code contains an example where a M25P05 SPI Flash memory is connected to the first slave select line on SPI bus 1. To enable support for this specific memory device, select “Memory Technology Devices / Self-contained MTD device drivers / Support for M25 SPI Flash”. To add support for another type of

SPI device, modify the example and configure support for the SPI device in question.

3.5.14 GRLib GRUSBHC

Support for GRUSBHC is enabled under “USB support / Support for Host-side USB”. Depending on the core's configuration, the applicable host controller drivers are “EHCI HCD (2.0) support” and “UHCI HCD (most Intel and VIA) support”.

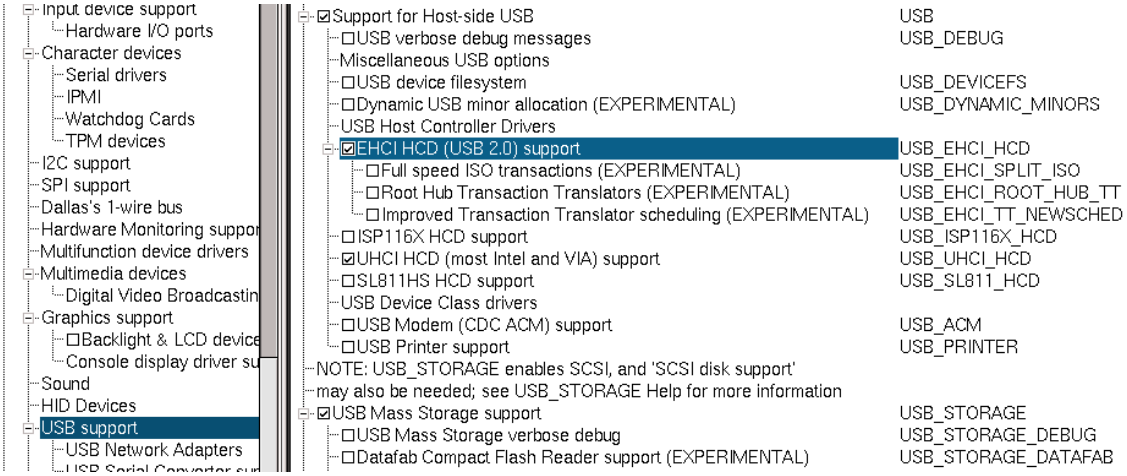

Illustration 3.9: Adding support for USB 2.0 host controller.

To enable support for USB storage devices, “SCSI / SCSI device support” and “SCSI / SCSI disk support” must first be enabled. USB connected human interface devices such as mice and keyboards are also supported. Enable “USB / USB Human Interface Device (full HID) support”, and INPUT_KEYBOARD and INPUT_MOUSE under “Input device support”.

3.5.15 GRLib GRVGA

GRLib SVGA controller can be used in Linux using the frame buffer video driver written for GRVGA. X-Windows and/or a frame buffer console can be run on top of the frame buffer driver. Normally one want mouse or at least a keyboard together with the graphical interface, GRPS2 can be used to connect keyboard and mouse devices.

One may need to consider bus bandwidth when selecting resolution and bit depth. The GRVGA controller will cause heavy bus loads for high resolution on slower buses. Each word in the frame buffer will be read 60 times on a system with 60 Hertz vertical refresh rate.

From the “Graphics support” menu one can enable the GRVGA frame buffer driver.

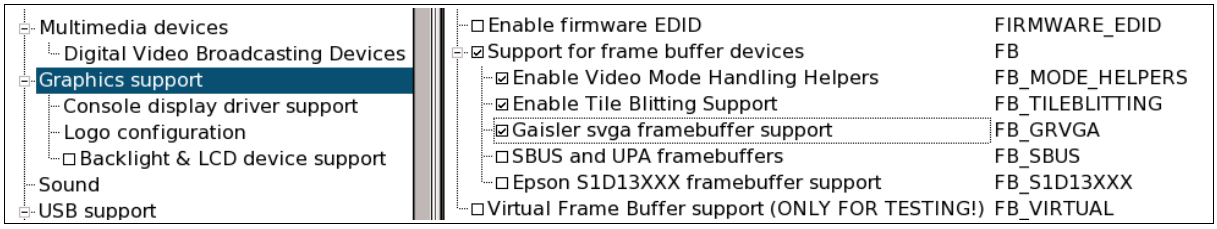

Illustration 3.10: Adding Frame Buffer driver.

When using the SVGA controller to provide console interface to the system the resolution, bit depth and other parameters can be set from the kernel parameters. Table 3.4 lists available arguments to the driver.

The Mem_size parameter can be calculated as resolution times bit depth in bytes. For example 614400 bytes video buffer makes a resolution of 640x480 8-bit, 640x480 16-bit and 800x600 8-bit resolution possible, however 800x600 16-bit would not fit into the memory as that would need 960000 bytes.

Order	Value	Custom/All	Description
0	video=grvga:	All	Needed to select frame buffer driver
1	Custom	All	Select one of the values to the left. When full control is needed specify the custom mode. When custom mode is selected arguments 2-10 are needed in addition to 11 and 12.
	1024x768@60		
	800x600@72		
	800x600@60		
	640x480@60		
2	Pixelclock	Custom	Pixelclock in ns.
3	xres	Custom	Horizontal resolution in pixels.
4	rmargin	Custom	Horizontal Front porch in pixels.
5	hsync_len	Custom	Horizontal Sync length in pixels.
6	lmargin	Custom	Horizontal Back porch in pixels.
7	yres	Custom	Vertical resolution in lines.
8	llmargin	Custom	Vertical Front porch in lines.
9	vsync_len	Custom	Vertical Sync length in lines.
10	umargin	Custom	Vertical Back porch in lines.
11	bit_per_pixel	All	Pixel depth in bits: 8,16,32-bits
12	Mem_size	All	Frame buffer memory size in bytes.

Table 3.4: GRVGA kernel parameter arguments

Below is an example how to configure the GRSVGA using 1024x768 resolution, 60Hz vertical refresh rate, 8-bits pixel depth and 800kb video buffer. The system console will be displayed on virtual terminal zero /dev/tty0 which is connected to the framebuffer */dev/fb0* instead of a serial terminal.

```
console=tty0 video=grvga:1024x768@60,8,786432
```

1024x768@60 is a predefined mode by the driver, however a custom mode can be entered, see the table above and kernel documentation for more details.

3.5.16 GRLib ATA Controller

Gaisler ATA controller can interface a hard disk or a compact flash card. Devices can be accessed from */dev/hda*. The ATACTRL can be selected from "ATA/.. support" menu.

The ATA controller depend upon the PCI subsystem to function correctly, PCI can be included from "General Machine Setup".

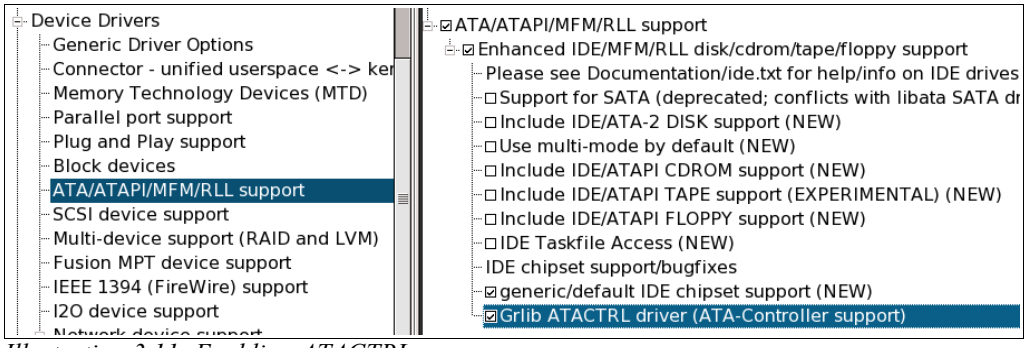

Illustration 3.11: Enabling ATACTRL.

Once the ATA controller is enabled the IDE devices that needs to be supported are to be selected, most commonly the IDE/ATA-2 DISK support is selected.

3.5.16.1 DMA Extension

The DMA extension of the GRLib ATA controller can be enabled by selecting BLK_DEV_IDEPCI, BLK_DEV_IDEDMA_PCI in addition to the options described for the standard ATA controller.

Note that enabling the DMA extension does **not** disable the standard ATA controller driver.

3.6 Configuring the 2.0.x kernel

From the graphical interface it is possible to configure the selected kernel by selecting "y" at "Customize Kernel Settings" and saving the new settings. After the main GUI has been closed and settings been saved a second GUI will appear that is specific for the kernel selected. The kernel configuration GUI for Linux 2.0 is shown below. Details on how to configure the kernel can be found in the linux-2.0.x/Documentation directory.

The following sections will be used to describe some of the Linux kernel's settings specific to LEON Linux.

3.6.1 LEON processor type

The configuration of the processor type is done separately for the Linux boot loader and the kernel itself. This is done to eliminate the kernel's dependency on SnapGear, LEON Linux can be used without SnapGear.

Certain hardware is only available for LEON3, their drivers will be invisible when LEON2 is selected.

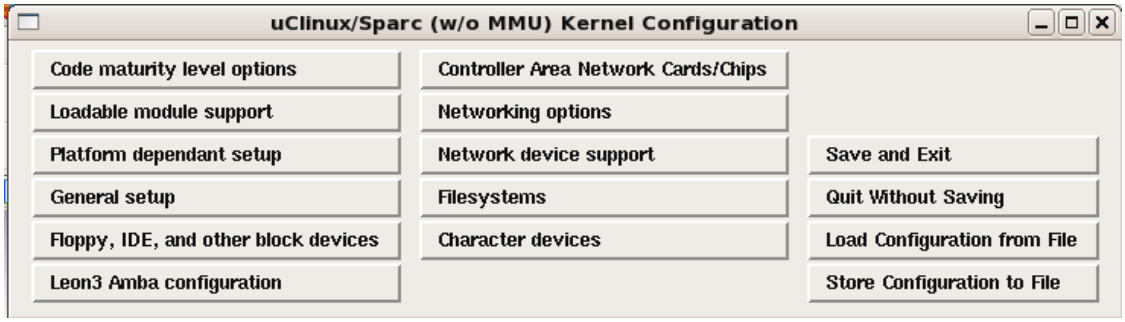

Illustration 3.12: Linux 2.0.x kernel configuration utility

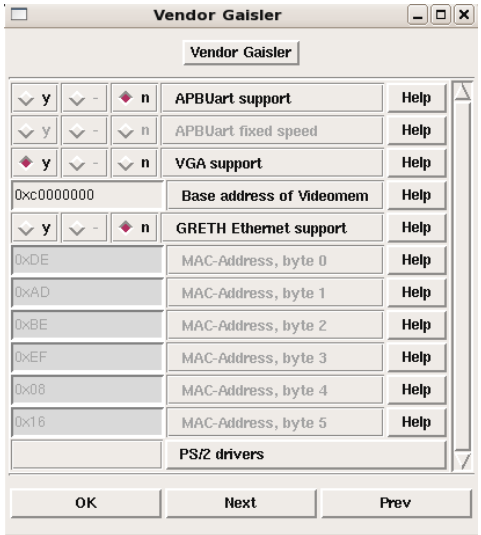

Illustration 3.13: LEON3 AMBA device configuration

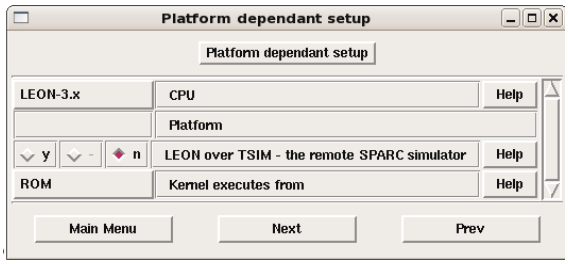

Illustration 3.14: Platform set up

3.6.2 GRLib APBUART (LEON3)

Select GRLIB_GAISLER_APBUART to include a UART driver for APBUART. One of the serial terminals can be set up as the system console via the kernel boot parameter, ex: console=ttyS0,38400 selects the first serial channel to act as the system console.

The serial terminals will be available by accessing /dev/ttySx.

3.6.3 LEON Serial (LEON2)

From “Character devices” sub menu the LEON2 UART driver can be enabled. It allows user space to communicate with an arbitrary protocol over the serial line terminal interface /dev/ttySx.

This driver uses hard coded addresses instead of probing the AMBA bus for Plug&Play information. This driver is intended for use with LEON2 only, LEON3 uses the AMBA Plug&Play information.

3.6.4 GRLib GRETH 10/100 Ethernet MAC

Select GRLIB_GAISLER_GRETH to enable the 10/100 Ethernet MAC.

The Ethernet MAC address of the GRETH MAC can be edited directly from the GUI. The address is made out of an unique 6 byte sequence. Byte 0 is the most significant byte. The Ethernet hardware address can be found by running */sbin/ifconfig* on a UNIX machine.

3.6.5 GRLib OpenCores 10/100 Ethernet MAC

OpenCores 10/100 Ethernet MAC modified for the AMBA bus included in GRLib can be used by enabling the driver under “Leon3 AMBA / Vendor Opencores“.

The Ethernet address must be set to an unique 6-byte network identifier.

The driver can force the Ethernet MAC to operate on a 100MHz signalling frequency by setting “Set MII to 100mb”.

3.6.6 SMC 91C111 10/100 Ethernet MAC

After enabling “Networking support” from the “General setup” sub menu the SMC driver can be enabled under “Network device support”. The base address, IRQ and Ethernet address is configurable. The Ethernet address must be unique on the network.

The base address and the IRQ number can be found by examining the hardware set up of the target board, typically found in the user manual or the schematics.

3.6.7 GRLib VGA text frame buffer support

The VGA text frame buffer make it possible to output text onto a standard monitor. The programming interface used to access the AMBA VGA graphics controller is called frame buffer. The frame buffer has hardware device drivers called frame buffer drivers, the Gaisler VGA controller has a frame buffer driver that can be included into the kernel from the AMBA configuration menu, but first frame buffer support needs to be enabled. Enable “Console support” and “Frame buffer” under “General setup”, this makes the “VGA support” able to select.

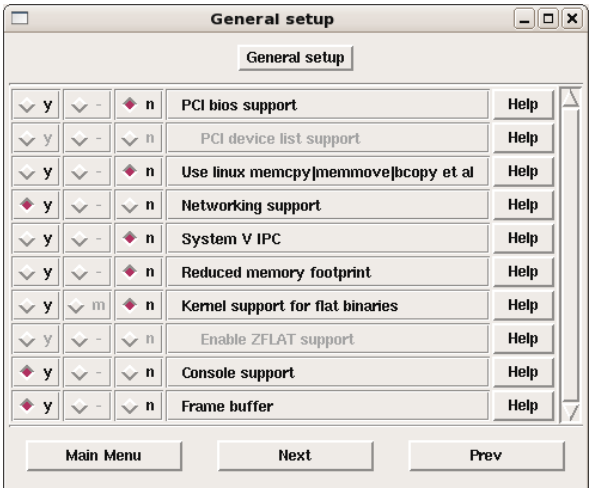

Illustration 3.15: General setup, console on frame buffer

3.6.8 GRLib GRPS2 PS/2 interface/keyboard

The PS/2 driver is dependent on the VGA driver, see the section above on how to include the VGA driver. The main purpose of the PS/2 driver is to provide the ability to interface a keyboard. Both the PS/2 driver and the PS/2 keyboard driver must be enabled to be able to use the keyboard. The keyboard supported is an AT keyboard.

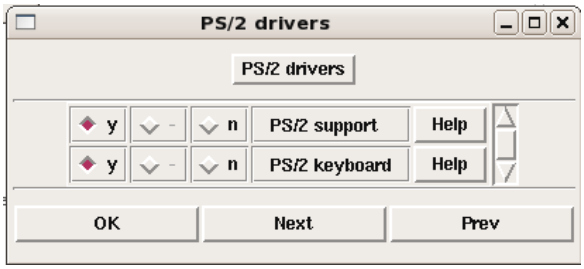

Illustration 3.16: AMBA PS/2 driver

3.7 Applications included in ROMFS

Apart from the Linux kernel the SnapGear RAD environment consist of applications in the snapgear-pxx/user directory that can be included into the root file system. The root file system is a read only file system that will be copied and decompressed into RAM during the boot process.

Custom applications can be easily integrated into the directory structure and SnapGear. The steps involving adding custom applications is described in a separate chapter.

From the main SnapGear GUI one can make a third GUI pop up after the main SnapGear GUI closes. This is similar to what was earlier described for the Linux kernel. One simply selects “y” at “Customize Vendor/User Settings” under “Kernel/Library/Defs Selection” and press “Quit and Save”.

BusyBox is a small footprint replacement for traditional UNIX core applications such as ls, find, mount etc. BusyBox is highly configurable and can be configured from the BusyBox sub menu as shown below.

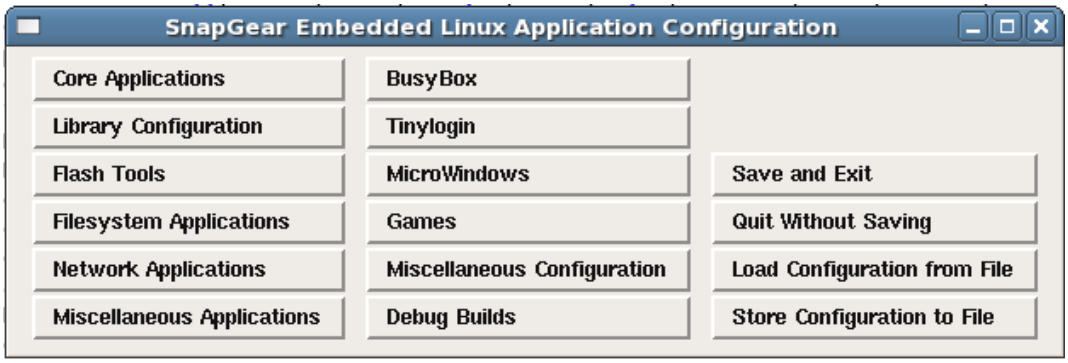

Illustration 3.17: SnapGear application configuration main menu

Since SnapGear supports both the 2.0.x and the 2.6.x kernels and some of the applications needs certain kernel interfaces, they may only be compiled for one of the kernels. A typical example of this is the flash support in Linux, from Linux 2.4 and onwards a new interface called MTD (memory technology devices) has been introduced. Linux 2.0 lacks the MTD interface and therefore cannot run applications that depend upon the MTD interface. Flash utilities can be found under "Flash Tools".

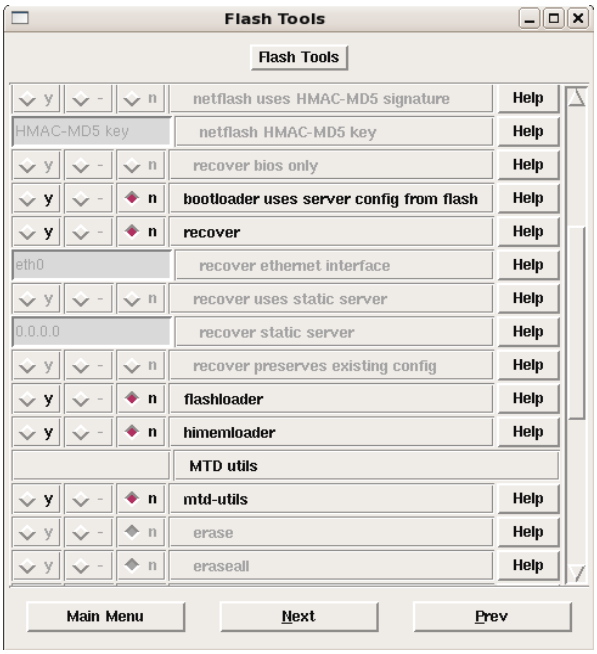

Illustration 3.19: Flash Tools, MTD utilities

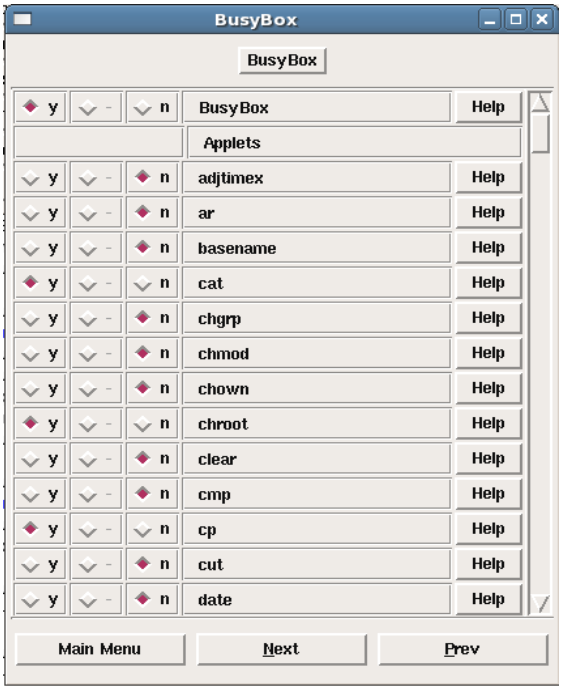

Illustration 3.18: BusyBox configuration

3.8 Template configurations

The "Template configurations" from the SnapGear main menu are provided as an source of examples. It contains prepared Linux kernel, boot loader and SnapGear application configurations. However it does not include setting and script files such as *inittab* and *rcS*.

Some of the prepared configuration files has been created for a certain template design. The template designs can be found in the *design* directory in GRLIB. These configuration files has been used to generate the images found in ftp://ftp.gaisler.com/gaisler.com/anonftp/linux/images.

It is also possible to add custom configurations easily by creating a directory under *vendor/gaisler/target/templates/config_dir* and putting the snapgear-pxx/.config into the config_dir named *vendor.config* and snapgear-pxx/linux-x.x.x/.config named *linux.config*. The configuration files will automatically be copied into their respective directory upon selection. The template name *config_dir* may not

contain '-' characters.

4 BUILDING SNAPGEAR

After configuring the kernel and the applications it is possible to compile and build the SnapGear LEON Linux distribution. There are two very important options passed to the build scripts, the FPU and SPARCV8 options found in “Gaisler/Leon2/3/MMU options”.

Disabling the FPU makes the compiler replace the FPU instructions with software routines that calculates the answer without needing a FPU. The compiler is run with the argument *-msoft-float*.

SPARC v8 processors have support for hardware integer multiplier/divider through the instructions MUL and DIV instructions whereas SPARC v7 hasn't. LEON is a highly configurable processor, it can be compiled with or without hardware integer multiplier support. To make the compiler generate code without MUL and DIV instruction select “n” for the v8 option. The compiler will generate code compatible with SPARC v7 if started with *-mcpu=v7*.

SnapGear is configured and compiled with:

```
make xconfig   # config GUI
make dep       # only Linux 2.0.x needs this
make           # compile kernel, libraries, boot loader, applications
               # and make images.
```

The resulting image produced during the build stage is put in snapgear-pxx/images.

Image	Function
image.dsu	RAM image, with partial boot loader $ grmon nb grmon> load image.dsu grmon> run
image.flashbz	Compressed Flash/PROM Image, with complete boot loader. $ grmon -nb grmon> flash erase 0 0x00300000 grmon> flash load image.flashbz grmon> run 0
image.tsim	TSIM LEON simulator image, if MMU is used TSIM must also support MMU. $ tsim-leon3 tsim> load image.tsim tsim> run

Table 4.1: Images available

The images can be downloaded and run using grmon as shown in table 4.1. Be sure to invoke grmon with the *-nb* option so that Linux can take care of traps instead of having grmon stop the execution.

5 ADDING CUSTOM APPLICATIONS

Custom applications can be added into the SnapGear projects in several ways, the simplest ways is to add the source code to the already prepared custom directory, *snapgear-pxx/user/custom*. It is also rather easy to modify the menu of the SnapGear Application menu and add a new application to the GUI. It also possible to copy a binary compiled outside of the SnapGear distribution folders. This chapter shows the simplest possible alternative, adding an application to the user/custom directory.

5.1 Creating an application

A simple application that print out the number of arguments it was invoked is saved to *user/custom/args.c*:

```
#include <stdio.h>

int main(int argc, char *argv[]){
  printf("%s: you passed %d arguments\n",argv[0],argc);
  return 0;
}
```

5.2 Setting up compilation directives

For the application to be compiled one must add it to *user/custom/Makefile*. The Makefile may be edited as follows.

Add these 4 lines:

```
EXEC9 = args
OBJS9 = args.o

$(EXEC9): $(OBJS9)
          $(CC) $(LDFLAGS) -o $@ $(OBJS9) $(LDLIBS$(LDLIBS_$@))
```

Add *$(EXEC9)* to the end of the *all* statement:

```
all: $(EXEC1)...$(EXEC5) $(EXEC6) $(EXEC8) $(EXEC9) INSMODEXE
```

5.3 Including application to file system

From the "Core Applications" it is possible to enable custom applications, the make utilities will enter the *user/custom* directory and compile it as described by *user/custom/Makefile*.

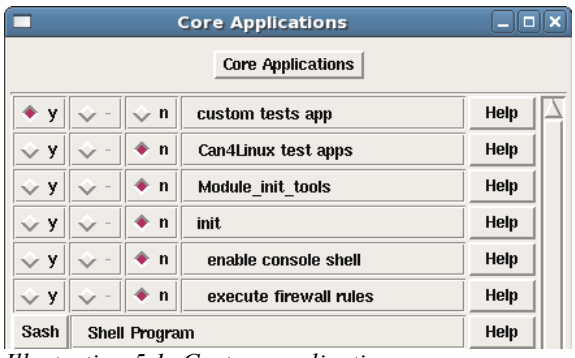

Illustration 5.1: Custom applications

6 DEBUGGING LINUX-2.6 AND APPLICATIONS

This section discusses different debugging methods used when debugging Linux 2.6.x userspace applications and the kernel itself.

GRMON is documented in detail in the GRMON manual available at *http://www.gaisler.com/doc/grmon.pdf.*

When debugging with GRMON the SPARC v8 Architecture manual may come in handy for instruction definition and the SPARC assembler conventions used by the C-compiler. The SPARC v8 manual is available at *http://gaisler.com/doc/sparcv8.pdf.*

GRLIB IP core documentation is very use full when debugging the kernel and writing new drivers, it can be found at *http://gaisler.com/products/grlib/grip.pdf.*

GDB documentation is available at *http://sourceware.org/gdb/documentation.*

6.1 Debugging symbols

In order to translate assembler instructions into C-code, debugging symbols and an application that can read debugging symbols are needed. The application *sparc-linux-objdump* part of the *binutils* package provided with the toolchain can provide us with information about compiled applications (ELF-binaries).

Debug symbols is made by GCC (*sparc-linux-gcc*) during compile time when at least one of the flags *-g, -g3*, *-ggdb, -gstabs* are given. The debug symbols enlarges the output binary, but the debug symbols can be removed - stripped - with the *sparc-linux-strip* utility prior to usage. An example of how the debugging information can be created, viewed and removed:

```
$ sparc-linux-gcc -g3 userapp.c -o userapp      # create app and syms
$ sparc-linux-objdump -S userapp > userapp.S    # deasm binary
$ sparc-linux-objdump -x userapp > userapp.x    # list sections
$ sparc-linux-strip -s userapp                  # remove syms
$ less userapp.S                                # view output
```

Debuggers can read the debug information directly from the binary making some of the steps above unnecessary.

The GDB server run on the target, later discussed, does not need debug information, to save space the binary is stripped with *sparc-linux-strip.*

Debugging information can be enabled in SnapGear from the "Application Configuration/Debug Builds/build debugable applications".

6.2 Debugging the kernel

In practice, hardware debug support of some kind is needed when debugging the kernel. It is possible to debug the kernel using various other methods such as inserting *printk* calls in the kernel code or using the */proc* filesystem. See the chapter "Debugging Techniques" in "Linux Device Drivers" for more information about debugging techniques. However these methods are rather time consuming and may change the behaviour and the execution path. Using a hardware controlling debugger gives the user a completely new way of controlling and monitoring the execution. Hardware debuggers such as GRMON vastly accelerates the development process. It is possible to read and manipulate both processor and core registers, walk the MMU page set up, view cache content, list previous instructions (instruction trace) just to list a few.

The kernel's addresses are static and can be known on before hand whereas multiple user space applications/threads may have the same virtual address making it impractical in many cases to use hardware debuggers. Using a software debugger like GDB run natively or a GDB server acting as a server for remote GDB connections is preferred when debugging userspace applications.

6.2.1 Configuring GRLIB for kernel debugging

It is assumed that hardware breakpoints/watchpoints and the instruction trace buffer have been enabled in the GRLIB *xconfig* during system configuration. Depending on the complexity of the problem to be debugged the number of breakpoints and the trace buffer length to select may vary. In the template design of the GRXC3S-1500 board the trace buffer can be enabled from "Processor/Debug Support Unit/Instruction trace buffer" and hardware breakpoints from "Processor/Integer Unit/Hardware watchpoints" in GRLIB *xconfig* (cd designs/leon3-grxc3s-1500; make xconfig).

GRMON can verify the debugging support available by running the command '*info sys*', below is a design with 2 hardware breakpoints (2 hwbp) and a 128 instruction deep trace buffer (itrace 128).

```
$ grmon -jtag
grmon> info sys
...
02.01:004   Gaisler Research  LEON3 Debug Support Unit (ver 0x1)
             ahb: 90000000 - a0000000
             AHB trace 128 lines, stack pointer 0x43fffff0
             CPU#0 win 8, hwbp 2, itrace 128, V8 mul/div, srmmu, lddel 1,
GRFPU-lite
                   icache 1 * 1 kbyte, 32 byte/line
                   dcache 1 * 1 kbyte, 32 byte/line
...
```

6.2.2 Using GRMON

GRMON can be used to debug the Linux kernel, it is primarily an assembler debugger but it is C-symbol aware. Symbols created by *gcc* when compiling C code with the *-g* flag are automatically loaded by GRMON after a successful *load file* command, but is also possible to load symbols manually by issuing the command *symbol file.* The symbols last loaded are matched first.

During the image creating process in SnapGear (make) the *image/* directory is populated with various images: *image, image.dis, image.dsu.* The *image.dsu* is to be loaded into RAM and run, *image* contain the debug information of the Linux kernel's virtual addresses and *image.dis* is a plain text file containing the disassembly of *image.*

Starting GRMON with the *-nb* flag is essential so that Linux can handle traps correctly. The trace buffer can be enabled by the command '*tm both*' and later listed by *inst.*

Breakpoints trigger when the processor fetches an instruction at the given breakpoint address. Hardware breakpoints are inserted with '*hbreak address*' or '*hbreak symbol*' . Inserting breakpoints with *break* instead *hbreak* causes GRMON to insert an instruction (ta 1). Hardware breakpoints is preferred when working with virtual addresses or debugging non writeable areas. GRMON stop the processor from executing further when a breakpoint is reached, the exact state can be observed and manipulated. Breakpoints triggers on addresses the processor executes, in this case virtual addresses.

The watchpoints implemented in GRLIB have support for stopping the processor when an address is being accessed by the software, all variants if the instructions *ld* or *st* are supported. As with hardware breakpoints watchpoints trigger on the address the processor accesses **before** MMU translation, thus virtual addresses is to be used when debugging the Linux kernel and its applications with GRMON. Watchpoints need hardware assistance to work, this cannot be done in software.

The architecture independent starting point of the kernel is at *start_kernel,* at this point the kernel is executing in virtual address space.

6.2.3 GRMON Example: debugging the Linux kernel

Below is an example of how the first call to *printk* can be debugged. By setting a hardware breakpoint on *printk* the processor halts after the *save* instruction has been executed. Viewing the INS registers one can inspect the arguments passed to the *printk* function. See SPARC v8 manual for SPARC calling conventions. From the *printk* C-prototype we know that the first argument must hold a pointer to a format string, thus *i0* register holds a pointer to a string.

The register content can be viewed by the command *reg*. From the output of *reg* one can see that *i0* is 0xF0227510. Listing the memory, with *vmem,* around the virtual address 0xF0227510 reveals that the argument passed to *printk* was "PROMLIB: Sun Boot Prom Version %d Revision %d".

A virtual address can easily be translated to a physical address by doing a "MMU page table walk" with the GRMON command *walk*. In the example below the virtual address 0xF0227510 is translated into 0x40227510 by the MMU. Listing the memory content, this time with *mem* instead of *vmem*, confirms that the memory content is the same for the physical address as the virtual address.

By looking at the previous 20 instructions of the trace buffer it is easy to see that the instruction *call* is executed two instructions before entering *printk.* As expected the call address is the address of *printk.* In the delay slot *or* is executed. Searching for 0xf0031e18 in *image.dis* tells us that the caller is *prom_init*, from the string passed to *printk* is seems reasonable.

As *printk* processes the string for output it must access the characters at some point or another, that point can be found be setting a watchpoint at a character address. To demonstrate the watchpoint functionality the processor is stopped when accessing the character S in the string. A watchpoint it set to 0xF0227519(S). As the execution continues it can be observed that the processor is stopped at the space character, the character just before S, this is because watchpoints must be aligned to a 32-bit boundary, GRMON does this for us.

The GRMON command *bt*, short for backtrace, show the current call history, from it one can see that *prom_init* called *printk* which called *vprintk* and so on.

```
$ grmon -jtag

grlib> tm both
combined instruction/AHB tracing

grlib> lo image.dsu
section: .stage2 at 0x40000000, size 10240 bytes
section: .vmlinux at 0x40004000, size 3670272 bytes
total size: 3680512 bytes (226.5 kbit/s)
read 5814 symbols
entry point: 0x40000000

grlib> symbol image
read 5805 symbols
entry point: 0xf0004000

grlib> hbreak printk

grlib> run
breakpoint 1  printk (0xf0031e1c)

grlib> reg

          INS       LOCALS      OUTS     GLOBALS
   0:  F0227510   F0399000   F02275E0   00000000
   1:  00000000   40000400   00000000   00000002
   2:  00000000   00000000   00000000   F31010E3
   3:  F0000C7C   00000000   00000140   F0000D94
   4:  00000000   00000000   00000000   F0237000
   5:  00000000   00000000   00000001   00000000
   6:  F000FF38   00000000   F000FED0   F000E000
   7:  F0266124   00000000   F02666BC   F0000C7C

 psr: F3401FE6   wim: 00000001   tbr: F0004050   y: 00000000
 pc:  f0031e1c  mov  %i0, %o0
 npc: f0031e20  st  %i1, [%fp + 0x48]

grlib> dis 0xf0031e10
 f0031e10  81c3e008   retl
 f0031e14  01000000   nop
 f0031e18  9de3bf98   save  %sp, -104, %sp
 f0031e1c  90100018   mov  %i0, %o0
 f0031e20  f227a048   st  %i1, [%fp + 0x48]
 f0031e24  f427a04c   st  %i2, [%fp + 0x4c]
 f0031e28  f627a050   st  %i3, [%fp + 0x50]
 f0031e2c  f827a054   st  %i4, [%fp + 0x54]
 f0031e30  fa27a058   st  %i5, [%fp + 0x58]
 f0031e34  40000004   call  0xf0031e44
 f0031e38  9207a048   add  %fp, 72, %o1
 f0031e3c  81c7e008   ret
 f0031e40  91e80008   restore  %o0, %o0
 f0031e44  9de3bf58   save  %sp, -168, %sp
 f0031e48  113c0e2d   sethi  %hi(0xf038b400), %o0
 f0031e4c  d2022000   ld  [%o0], %o1

grlib> vmem 0xF0227510
 40227510  50524f4d  4c49423a  2053756e  20426f6f    PROMLIB: Sun Boo
 40227520  74205072  6f6d2056  65727369  6f6e2025    t Prom Version %
 40227530  64205265  76697369  6f6e2025  640a0000    d Revision %d...
 40227540  61766169  6c61626c  65000000  00000000    available.......

grlib> walk 0xF0227510
Tablewalk: (f0)(8)(27)
+ctx(0):40002000 ctx->4000241
+region(f0):400027c0 region->400007e
 segment->400007e
 page->400007e
= 40227510(pte:400007e)
```

```
grlib> mem 0x40227510
 40227510  50524f4d  4c49423a  2053756e  20426f6f    PROMLIB: Sun Boo
 40227520  74205072  6f6d2056  65727369  6f6e2025    t Prom Version %
 40227530  64205265  76697369  6f6e2025  640a0000    d Revision %d...
 40227540  61766169  6c61626c  65000000  00000000    available.......

grlib> inst 20
   time      address   instruction                 result
    9303794  f02666d4  cmp  %o0                    [00000000]
    9303806  f02666d8  bne  0xf0266700             [00000000]
    9303807  f02666dc  mov  320, %o3               [00000140]
    9303808  f02666e0  sethi  %hi(0xf0399000), %o0 [f0399000]
    9303819  f02666e4  ld  [%o0 + 0x3c0], %o2      [00000000]
    9303820  f02666e8  sethi  %hi(0xf0227400), %o0 [f0227400]
    9303833  f02666ec  orcc  %o2, %o1              [00000000]
    9303845  f02666f0  be  0xf0266724              [00000000]
    9303850  f02666f4  or  %o0, 0x1e0, %o0         [f02275e0]
    9303851  f0266724  nop                         [00000000]
    9303861  f0266728  ret                         [f0266728]
    9303862  f026672c  restore                     [00000000]
    9303900  f0266114  ld  [%l0 + 0x34c], %o0      [f0000c7c]
    9303925  f0266118  ld  [%o0 + 0x4], %o1        [00000000]
    9303944  f026611c  ld  [%l1 + 0x354], %o2      [00000000]
    9303955  f0266120  sethi  %hi(0xf0227400), %o0 [f0227400]
    9303965  f0266124  call  0xf0031e18            [f0266124]
    9303973  f0266128  or  %o0, 0x110, %o0         [f0227510]
    9303983  f0031e18  save  %sp, -104, %sp        [f000fed0]
    9303990  f0031e1c  mov  %i0, %o0               [trapped]

grlib> watch 0xF0227519

grlib> cont
watchpoint 1  vsnprintf + 0x68 (0xf010edb0)

grlib> reg
          INS       LOCALS      OUTS     GLOBALS
   0:  F038B000   00000000   0000003A   00000000
   1:  00000400   F038B3F8   0000003A   00000002
   2:  F0227518   00000000   FFFFFFFF   F3401FE5
   3:  F000FF80   F038AFF8   F00001D4   F0000D94
   4:  00000000   00000000   00000000   F0237000
   5:  00000000   00000000   00000000   00000000
   6:  F000FDC0   00000000   F000FD48   F000E000
   7:  F010F348   00000000   F01109B8   F0000C7C

 psr: F3901FE3   wim: 00000001   tbr: F0004050   y: 00000000
 pc:  f010edb0  ldub  [%i2], %o1
 npc: f010edb4  sll  %o1, 24, %o0

grlib> vmem 0xF0227518 0x10
 40227518  2053756e  20426f6f  74205072  6f6d2056     Sun Boot Prom V
```

6.3 Debugging userspace applications

Debugging user space applications is a lot different from kernel debugging. One can still use GRMON the same way as for kernel debugging but is only effective in simpler debugging cases. When debugging userspace applications a native GDB debugger, executing on the LEON target, may be used to debug applications using the serial console for input. This is similar to debugging PC applications and not described here. The CPU may be heavily loaded since GDB share the CPU with the application, also application sources and debug binaries are required at the target when debugging natively.

For targets having a serial port or a network connection it is possible to debug Linux applications remote. The debugging interface is presented on a PC unloading the target CPU. The most common solution is to use a GDB TCP/IP server (gdbserver) exporting control to a remote PC running GDB (sparc-linux-gdb) compiled for target binaries (SPARC).

GRMON must be started with the flag *-nswb* in order to instruct GRMON not to interfere with the debugging process or else software breakpoints result in GRMON taking over CPU, beware this only needed when debugging userspace applications not when debugging the kernel.

6.3.1 Setting up a debugging environment

Together with the sparc-linux toolchain (version 1.0.1 and onwards) comes *bin/sparc-linux-gdb* and *bin/sparc-linux-insight*. *sparc-linux-gdb* is GDB built for PC Linux able to read/debug SPARC Linux binaries.

The Data Display Debugger (DDD) is normally available in for download for any distribution. DDD is optional as GDB can be used effectively in text mode.

The GDB TCP/IP server can be built within SnapGear by enabling it from the Application Configuration Utility under "Miscellaneous Applications/gdbserver".

With GRMON version 1.1.23 the flag *-nswb* was added, it clears bit 3 (BS) in the DSU control register so that GRMON won't interfere with the software breakpoints (ta 0x1).

6.3.2 GDB introduction

GDB is a debugger able to debug multiple languages including C and SPARC assembly. In this text GDB is used to access a GDB server running on the target board debugging. GDB needs

- Binaries of the application and optionally the used libraries
- Sources used to compile the binaries
- The IP number of the target, can be found with "`target$ /sbin/ifconfig`"
- *gdbserver* running on the target (part of SnapGear)
- The *gdbserver* also need the binary but with or without debug information. Debug information is not needed by the GDB server.

depending on what to debug. The libraries and their source code may not be needed. When debugging LibC the sources and libraries are of course needed.

See the previous section "Debugging symbols" on how to create binaries with debugging information.

6.3.2.1 Adding additional source search paths

Sometimes GDB can not list the C-code, that may be due to missing debug information (not even searching for C-code) or it can not find the source files mentioned in the debug information. It is not recommended to change the source files after the debug binary has been created.

If the binary is compiled on the same computer running *sparc-linux-gdb* one need not to add additional paths normally. The path information is coded into the binary, however when running GDB on a different computer than the build host the path information is invalid and it may be needed to add additional search paths. Search paths can be added with the '*-d dir1:dir2:dir3*' flag passed to GDB.

6.3.3 Starting GDB server on target

Assuming that the compilation of *gdbserver* has been successful and that GRMON has been started with the flag *-nswb* the GDB server can be started with the following command at the target:

```
target$ gdbserver :1234 app arg1 arg2
```

Where 1234 is the TCP/IP port, *app* is the application to be debugged and *arg1* and *arg2* are the arguments to *app*.

Note: The debugged application does not have to include any debugging information. It can be stripped.

6.3.4 Connecting with GDB to gdbserver

Below is an example of how to start GDB on the PC and connecting to the target, the target board IP address is assumed to be 192.168.0.52, the TCP port 1234.

```
pc$ cd sources
pc$ sparc-linux-gdb ./app_with_debug_info
(gdb) target remote 192.168.0.52:1234
```

6.3.5 GDB example usage

Configuring SnapGear with glibc-from-compiler, linux-2.6.21.1, with gdbserver,

```
pc$ make xconfig
```

Building SnapGear,

```
pc$ make
```

The binary application is recompiled manually with debugging symbols,

```
pc$ file romfs/bin/testsin
romfs/bin/testsin: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
dynamically linked (uses shared libs), stripped

pc$ cd user/custom
pc$ sparc-linux-gcc -lm -g3 testsin.c -o testsin-g
pc$ file testsin-g
testsin-g: ELF 32-bit MSB executable, SPARC, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
```

The serial console is connected to a terminal emulator such as *minicom* or *hyper terminal* with 38400baud 8N1.

Running the image *images/image.dsu* from RAM on target LEON board using GRMON,

```
pc$ grmon -jtag -nb -nswb
grmon> lo images/images.dsu
grmon> run
```

On the terminal emulator connected to LEON Linux target,

```
target$ gdbserver :1223 /bin/testsin
 Process /bin/testsin created; pid = 28
```

Starting GDB on the PC and testing that GDB finds the sources,

```
sparc-linux-gdb testsin-g
GNU gdb 6.4.0.20051202-cvs
Copyright 2005 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB.  Type "show warranty" for
details.

This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-
linux"...

(gdb) list
1       #include <math.h>
2       #include <stdlib.h>
3
4       int main(int argc,char **argv) {
5         double i = 0.00;
6         int j = 0;
7         printf("testsin:\n");
8         while (i < 100.0) {
9           double v = sin(i);
10          printf("float:sin(%f)=%f\n",i,v);
```

Connecting GDB to the remote *gdbserver,*

```
(gdb) tar rem 192.168.0.80:1223
Remote debugging using 192.168.0.80:1223
0x50001c20 in ?? ()
```

Setting breakpoints,

```
(gdb) break 7
 Breakpoint 1 at 0x10440: file testsin.c, line 7.

(gdb) break 10
 Breakpoint 2 at 0x10494: file testsin.c, line 10.
```

Running application with *cont* instead of *run* since application already has been started and paused on the target side,

```
(gdb) cont
 Continuing.

 Breakpoint 1, main (argc=1, argv=0xefbd1ed4) at testsin.c:7
 7         printf("testsin:\n");
```

Stepping one C-line with *next*, as the *printf* call is executed the output appears on the target console,

```
(gdb) next
  8         while (i < 100.0) {
```

Running the loop four times,

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xefbd1ed4) at testsin.c:10
10          printf("float:sin(%f)=%f\n",i,v);

(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xefbd1ed4) at testsin.c:10
10          printf("float:sin(%f)=%f\n",i,v);

(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xefbd1ed4) at testsin.c:10
10          printf("float:sin(%f)=%f\n",i,v);

(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xefbd1ed4) at testsin.c:10
10          printf("float:sin(%f)=%f\n",i,v);

(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xefbd1ed4) at testsin.c:10
10          printf("float:sin(%f)=%f\n",i,v);

(gdb) print i
$1 = 5
```

The output on the LEON target terminal is now,

```
Sash command shell (version 1.1.1)

/> gdbserver :1223 /bin/testsin
Process /bin/testsin created; pid = 26
Remote debugging using :1223
testsin:
float:sin(0.000000)=0.000000
float:sin(1.000000)=0.841471
float:sin(2.000000)=0.909297
float:sin(3.000000)=0.141120
float:sin(4.000000)=-0.756802
```

One can change the execution flow by manipulating variables within the loop. The integer i set to 90, the breakpoints are removed and the application are continued to is endpoint,

```
(gdb) set variable i=90
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) cont
Continuing.

Program exited normally.
```

Finally the LEON target's console output was,

```
Sash command shell (version 1.1.1)
/> gdbserver :1223 /bin/testsin
Process /bin/testsin created; pid = 26
Remote debugging using :1223
testsin:
float:sin(0.000000)=0.000000
float:sin(1.000000)=0.841471
float:sin(2.000000)=0.909297
float:sin(3.000000)=0.141120
float:sin(4.000000)=-0.756802
float:sin(90.000000)=-0.958924
float:sin(91.000000)=0.105988
float:sin(92.000000)=-0.779466
float:sin(93.000000)=-0.948282
float:sin(94.000000)=-0.245252
float:sin(95.000000)=0.683262
float:sin(96.000000)=0.983588
float:sin(97.000000)=0.379608
float:sin(98.000000)=-0.573382
float:sin(99.000000)=-0.999207
exit testsin

Child exited with retcode = 0

Child exited with status 0
GDBserver exiting
/>
```

6.3.6 DDD and GDB

DDD is a graphical frontend run alongside with GDB. DDD launches GDB and provides us with direct access to the GDB console and the ability to display information about the debugged application. DDD is compiled for PC Linux and uses a socket protocol to connect to the local GDB (sparc-linux-gdb). GDB is responsible for SPARC specific operations and therefore DDD needs not to be recompiled for SPARC binaries.

```
pc$ ddd --debugger sparc-linux-gdb app_debug
 (ddd-gdb) target remote target:port
 (ddd-gdb) b main
 (ddd-gdb) cont
```

Most Linux distributions distribute DDD as a precompiled binary package, in many cases it is installed by default.

See *http://www.gnu.org/software/ddd* for more information about DDD.

6.3.7 Insight

Insight comes with version 1.0.1 or later of sparc-linux toolchains. Insight is a graphical debugging interface on top of GDB. It is started by running *sparc-linux-insight.*

See *http://sourceware.org/insight* for more information on how to use insight.

6.4 Using NFS to simplify application updates

During development one i often needed to update the application and rerun it. Using the standard procedure is very inefficient, rebuild the image, reprogram the flash and reboot. With GRMON the flash i not needed to be updated, the image can be run directly from RAM saving a great deal of time. However, there are occasions where even rebooting may be troublesome. Sharing the files over NFS may be a good alternative to rebooting and rebuilding all SnapGear. Only the development files/binaries need to updated, not all SnapGear.

Perhaps the easiest way is to create a script file in the SnapGear image which mounts NFS after boot, yet another way is to edit the *rcS* file in *vendors/leon3mmu/rcS* to mount automatically on start up.

See your distribution manual how to set up an NFS sharer, normally you edit */etc/exports* and run '*exportfs -r*' . The same problem is faced in the chapter "Root Filesystem over Ethernet using NFS".

Good practice is to mount the exported directory on to the server PC to verify that the NFS share is set up correctly (even if this works it may be wrong). After network and NFS server has been set up on the target one can simply mount the NFS share into an empty local directory with:

```
target$ mkdir /nfs
target$ mount -t nfs -o rw,nolock workstation:/export/leonshare
```

6.5 Console output when debugging

The console is of great help when debugging applications, but often it limits the execution speed or put in another way the console can not output enough information. There are ways to speed up the console output, two techniques are discussed here.

6.5.1 Redirecting output to NFS share

The console output can be redirected to a file by the shell. The shell simply connects the application's *stdout* to a file. For this to work the shell needs to support redirection, only tiny shells doesn't support redirection. It is also possible to redirect *stderr* to file. See '*man stdin*' and '*man bash*' for more info. Embedded storage is often limited in size and speed and would fill up the flash device rapidly. Instead a NFS server can be used to place the execution log over the network.

```
$ mount -t nfs -rw logserver:/export/logs /nfslogs
$ app > 2>&1 /nfslogs/out_10sep2007.log
```

6.5.2 TELNET over TCP/IP network

Logging onto the target with *telnet* is an easier solution, all that has to be done is to set up a telnet server on the target. A *telnet* server (telnetd) is available in SnapGear from the SnapGear Application configuration utility under Network Applications.

```
pc$ telnet target
username: root
pass: *****


Welcome to SPARC LEON3 Linux
----------------------------
root@target# app
```

7 PS/2 KEYBOARD AND VGA CONSOLE

This is an example of how to configure the Linux 2.6.x kernel, set up *init* and it's settings file *inittab,* the boot loader's parameters, and the kernel command line.

The software can be run on a GR-XC3S-1500 Spartan-1500 board available at www.gaisler.com.

7.1 Hardware configuration

The hardware set up is based on the template design of GR-XC3S-1500 and modified as in table 7.1.

Hardware / Controller	Function
LEON3 with MMU, HW MUL/DIV and FPU	SPARC v8 Processor with memory protection logic and floating point unit.
GRVGA	SVGA controller
GRPS2	PS/2 controller connected to keyboard

Table 7.1: Hardware in addition to GR-XC3S-1500 template design

7.2 Configuring the boot loader and main SnapGear options

Start SnapGear main GUI and configure as in table 7.2:

```
$ make xconfig
```

Name	Value
Vendor / Product Selection	
Vendor	Gaisler
Gaisler Product	Leon3mmu
Gaisler Leon2/3/mmu options	
SPARC v8	Yes
FPU support	Yes
Clock frequency	40MHz
Baudrate	38400
In memory root filesystem	initramfs
Kernel command line	console=tty0 video=grvga:640x480@60,8,307200
Kernel/Library/Defaults Section	
Kernel Version	Linux 2.6.x
Libc Version	Glibc-from-compiler
Customize Kernel Settings	Yes
Customize Vendor/User Settings	Yes

Table 7.2: Boot loader configuration

7.3 Configuring the Linux kernel

Apart from the default configuration support for frame buffer on Gaisler VGA controller and input driver for Gaisler PS/2 controller is enabled, as in table 7.3.

Name	Function
Graphics Support	
FB	Frame buffer sub system
FB_GRVGA	Frame buffer driver for GRVGA controller
FRAMEBUFFER_CONSOLE	Frame buffer console support
FONTS	Compiled in fonts
FONT_8x16	8x16 Font support
LOGO	Add boot logo support
LOGO_LINUX_CLUT224	Include a coloured Linux logo
Input devices	
SERIO_LEON3	Driver for Gaisler PS/2 controller
SERIO_LEON3_KEYB_IRQ	The IRQ number of assigned to the PS/2 controller connected to the keyboard
INPUT_KEYBOARD	Keyboard sub system
KEYBOARD_ATKBD	AT-keyboard driver

Table 7.3: Kernel configuration

7.4 Configuring SnapGear Applications

Below is a description of how the SnapGear ROMFS can be configured:

Name	Function
BusyBox	
Init	First application that gets started after boot
Init: use inittab	Make init read /etc/inittab for settings

Table 7.4: SnapGear configuration

7.5 Building the kernel and applications

Building the kernel, libraries and applications can be done as follows:

```
$ make
```

7.6 Setting up /etc/inittab

Init reads the */etc/inittab,* line after line at start up and launches the /etc/rc.sh system initialization script and spawns applications on each console as set up. *Inittab* can be created by typing:

```
$ cat << EOM > romfs/etc/inittab

# System initialization script
::sysinit:/etc/rc.sh

# Make serial terminal have a console
ttyS0::respawn:/bin/sh

# Make consoles appear via frame buffer
tty1::respawn:/bin/sh
tty2::respawn:-/bin/sh
tty3::askfirst:-/bin/sh

EOM
```

rc.sh can be typed in as follows:

```
$ cat << EOM > romfs/etc/rc.sh

#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devpts devpts /dev/pts

hostname sparky
/sbin/ifconfig lo up 127.0.0.1 netmask 255.0.0.0
route add 127.0.0.1 dev lo

EOM
```

For the new script to be able to run one must add execution permission to it:

```
$ chmod +x romfs/etc/rc.sh
```

7.7 Building again with inittab and rc.sh

Since changes has been made only to the file system one can rebuild the images without having to recompile kernel, libraries and applications.

```
$ make image
```

After building the images the image can be found at *images/image.dsu*. Note that *image.flashbz* does not function correctly since the boot loader's SDRAM and flash settings hasn't been set up properly.

7.8 Running on hardware

The far most simplest method of testing the image is to run it using grmon as follows. Depending on what debug interface is available the parameters to grmon may differ, see grmon documentation for details:

```
$ cd images
$ grmon -jtag -nb

grmon> load image.dsu
grmon> run
```

When everything is working as planned the connected monitor will first display a small penguin, kernel messages and then launch a shell on tty1, tty2 and the serial terminal (ttyS0). It is possible to switch between tty1-3 by pressing ALT+F1/F2/F3 at the connected PS/2 keyboard.

8 ROOT FILE SYSTEM OVER ETHERNET USING NFS

Often during development it is favourable to use NFS as root file system as it is a tedious process reprogramming the flash each time a change is made, and also there is no need in rebuilding the romfs image. Settings can be preserved after system reboot.

Setting up root over NFS involves two steps, installing the NFS server on the PC and configuring the LEON Linux image. The NFS server is assumed to be installed and therefore not described other than with an example entry applied to */etc/exports*, it make the NFS server export the root file system to the LEON board.

8.1 Setting up NFS server on PC

As previously mentioned the NFS server is assumed already to be installed, however an entry to */etc/exports* sharing the root file system should be added similar to what described. Adding the following line to the /etc/exports makes the server share the /export/rootfs directory. The NFS server is forced to reread the settings file by invoking “exportfs -r”.

```
/export/rootfs 192.168.0.0/255.255.255.0 (rw,async,no_root_squash,insecure)
```

In this example we export /export/rootfs which is a SnapGear ROMFS file system. The directory is exported to any computer using the IP address 192.168.0.X.

```
$ su
# mkdir /export
# cp -rpd ~daniel/snapgear-pxx/romfs /export/rootfs
# echo “/export/rootfs 192.168.0.0/255.255.255.0
(rw,async,no_root_squash,insecure)” >>/tmp/exports
# exportfs -r
# exit
$
```

It is also possible to base the root file system on other distributions for example on splack or creating one from scratch. A common source of failure is that the initialization scripts, run during boot, assumes that the file system is of ext2 or ext3 type, but it's not. This can usually easily be removed.

8.2 Configuring the boot loader and main SnapGear options

The important thing when running the root file system over NFS is to set up the kernel boot parameter to point to the NFS share to be used as root and the network settings such as IP address, netmask and gateway.

To avoid building the image with the unused ROMFS root file system the “In-memory root filesystem” option from the “Gaisler Leon2/3/mmu options” is set to “none”. The settings in the SnapGear application GUI does not effect the build process once “In-memory root filesystem” is set to “none”.

An example configuration is outlined in table 8.1, the PC NFS server has the IP address 192.168.0.20 and the target LEON board is configured to 192.168.0.203.

Name	Value
Vendor / Product Selection	
Vendor	Gaisler
Gaisler Product	Leon3mmu
Gaisler Leon2/3/mmu options	
SPARC v8	Yes
FPU support	Yes
Clock frequency	40MHz
Baudrate	38400
In-memory root filesystem	NONE
Kernel command line	console=ttyS0,38400 root=/dev/nfs nfsroot=192.168.0.20:/export/rootfs,nfsvers=3 ip=192.168.0.203:192.168.0.20:192.168.0.1:255.255.255.0:grxc3s_daniel:eth0:
Kernel/Library/Defaults Section	
Kernel Version	Linux 2.6.x
Customize Kernel Settings	Yes

Table 8.1: SnapGear main configuration

8.3 Configuring the Linux kernel

In addition to the default settings of the Linux 2.6.x kernel add the features described by table 8.2.

Name	Function
Networking	
NET	Networking support
INET	TCP/IP protocol
IP_PNP	IP settings can be set from kernel command line
UNIX	UNIX domain sockets, needed by Debian's logging facilities.
Device Drivers / Network device support	
NETDEVICES	Enables the network device driver interface
General machine setup / Grlib: AMBA / Gaisler	
GRLIB_GAISLER_GRETH	Add Gaisler Ethernet 10/100/1000 driver. Select a unique Ethernet address in the MSB and LSB fields.
Device Drivers / Block Devices	
BLK_DEV_INITRD	**Disable** ROMFS RAM root file system
File systems / Network file systems	
NFS_FS	Network file system support
NFS_V3	Support for version 3 of the NFS protocol
ROOT_NFS	Add support for root file system over NFS

Table 8.2: Kernel configuration for NFS root FS

8.4 Building kernel and boot loader

As previously described,

```
$ make
```

The image will be available for download to the target board from images/image.dsu.

8.5 Running on hardware

The far most simplest method of testing the image is to run it using grmon as follows. Depending on what debug interface is available the parameters to grmon may differ, see grmon documentation for details:

```
$ cd images
$ grmon -jtag -nb

grmon> load image.dsu
grmon> run
```

The NFS share can be tested by changing or adding files at the PC side and watch as they appear on the LEON Linux target.

9 ROOT FILE SYSTEM OVER ETHERNET USING ATA OVER ETHERNET

A network root filesystem can also be provided using ATA-Over-Ethernet on Linux 2.6. This setup requires two steps: creating an ATAoE server that exports a block device containing the root filesystem, and configuring the LEON to boot using the exported ATAoE root block device.

ATAoE block devices may be formatted with an arbitrary filesystem. For example, well-supported Linux filesystems such as ext3 or xfs may be used. The ATAoE server must be on the same Ethernet Local Area Network as the LEON.

9.1 Setting up ATAoE Server

The ATAoE server, *vblade*, can be run on any supported Unix system, including Linux and FreeBSD. Vblade can be obtained at http://aoetools.sourceforge.net .Installation documentation is included with vblade, although typing 'make' followed by 'make install' (as root) should be sufficient for Linux systems.

With vblade installed, the next step is to configure the block device that will be exported by vblade. The block device may be in one of two formats: a flat file or an actual block device. We will first show how to configure a block device for vblade, as this is the easier of the two options. This example formats the block device /dev/hdb using ext3, and then copies a SnapGear root filesystem to /dev/hdb.

```
# mke2fs -j /dev/hdb
# mount /dev/hdb /mnt/tmp
# cp -rpd ~daniel/snapgear-pxx/romfs /mnt/tmp
# umount /mnt/tmp
```

Now we will show how to configure a flat file to be used as a vblade-exported block device. In this example, we create a 4GB flat file /rootfs.img formatted with ext3 containing a SnapGear root filesystem.

```
# dd if=/dev/zero of=/rootfs.img bs=1024 count=4000000
# losetup /dev/loop0 /rootfs.img
# mount /dev/loop0 /mnt/tmp
# cp -rpd ~daniel/snapgear-pxx/romfs /mnt/tmp
# umount /mnt/tmp
# losetup -d /dev/loop0
```

Make sure that the vblade-exported root block device has been unmounted (and in the case of a flat file, that losetup -d has detached the file from any loopback devices) before running vblade. Now run vblade:

```
# vbladed eth0 0 0 eth0 /path/to/root-device
```

In this example, vblade is run with shelf 0 slot 0 over Ethernet network interface eth0, exporting the device at /path/to/root-device. To use the devices set up by our examples, we could replace /path/to/root-device with /dev/hdb or /rootfs.img, respectively. The shelf and slot numbers are used for naming purposes – they allow the server to run multiple vblades. As long as each block device is exported with a different shelf or slot number, the client can select which of the exported block devices it wishes to access. In our examples we will use shelf 0 slot 0 as we are not using multiple vblade servers.

The ATAoE server is now operational and its output will be sent via syslog to /var/log/messages. Note that because vblade operates over Ethernet and not TCP/IP, it does not bind to a port, nor is it accessible from any machine outside of the local LAN. For more information on ATA over Ethernet, see linux-2.6.x/Documentation/aoe/aoe.txt.

9.2 Configuring the boot loader and main SnapGear options

To mount a root filesystem using ATAoE, we must use Linux 2.6.x with an initramfs initial root filesystem. This filesystem should contain only the klibc utility, kinit. This utility will automatically parse the options we pass to the Linux kernel, mount the ATAoE block device, and switch the root filesystem over to the filesystem on the ATAoE mount.

An example configuration is outlined in the following table. The ATAoE server has slot and shelf number 0, and the target LEON board is configured to 192.168.0.203. Note that the ip= option may be omitted, as ATAoE requires only Ethernet and will function even if the LEON does not have a valid IP address. However, LEON system using ATAoE typically enable TCP/IP.

Name	**Value**
Vendor / Product Selection	
Vendor	Gaisler
Gaisler Product	Leon3mmu
Gaisler Leon2/3/mmu options	
SPARC v8	Yes
FPU support	Yes
Clock frequency	40MHz
Baudrate	38400
In-memory root filesystem	Initramfs
Init pathname	/bin/kinit
Kernel command line	console=ttyS0,38400 root=/dev/etherd/e0.0 ip=192.168.0.203:192.168.0.20:192.168.0.1:255.255.255.0:grxc3s_daniel:eth0:
Kernel/Library/Defaults Section	
Kernel Version	Linux 2.6.x
Customize Kernel Settings	Yes
Customize Vendor/User Settings	Yes
Libc version	None

9.3 Configuring the Linux kernel

In addition to the default settings of the Linux 2.6.x kernel add the features described by the following table.

Name	Function
Networking	
NET	Networking support
INET	TCP/IP protocol
IP_PNP	**Disable** IP PnP – kinit will provide this functionality for us
Device Drivers / Network device support	
NETDEVICES	Enables the network device driver interface
General machine setup / Grlib: AMBA / Gaisler	
GRLIB_GAISLER_GRETH	Add Gaisler Ethernet 10/100/1000 driver. Select a unique Ethernet address in the MSB and LSB fields.
Device Drivers / Block Devices	
BLK_DEV_INITRD	**Disable** INITRD support
ATA_OVER_ETH	Add ATA Over Ethernet block device support
File systems / Network file systems	
EXT3_FS	Ext3 journalling filesystem support

9.4 Configuring the vendor/user applications

At a bare minimum, klibc/kinit support must be enabled for ATAoE root to work. Furthermore, we recommend disabling all other applications, as there will be no need for them. This is because kinit immediately switches the root filesystem to the ATAoE block device at startup, and does not invoke or require any other applications or libraries. Configuring a root filesystem with only klibc/kinit, while not strictly required, will result in a much shorter build time and a much smaller initial root filesystem.

Name	Function
Core Applications	
Custom tests app	**Disable** custom tests app
Shell Program	Other
Network Applications	
arp	**Disable** arp
portmap	**Disable** portmap
tcpd	**Disable** tcpd
Busybox	
Busybox	**Disable** Busybox
Miscellaneous Configuration	
RAMFS Image	None
klibc	
build klibc	Enable klibc support
kinit	Enable kinit
Statically link all binaries	(Optional) Should result in a slightly smaller size for kinit – there's only one binary so static linking actually doesn't waste space in this case

9.5 Building kernel, boot loader, and kinit

As previously described,

```
$ make
```

The image will be available for download to the target board from images/image.dsu.

9.6 Running on hardware

The simplest method of testing the image is to run it using grmon as follows. Depending on what debug interface is available the parameters to grmon may differ, see grmon documentation for details:

```
$ cd images
$ grmon -jtag -nb

grmon> load image.dsu
grmon> run
```

10 RUNNING GRLINUX/SPLACK FROM AN ATA HARD DISK

Splack is a distribution based on the SPARC version of slackware, it is prepared especially for demonstrating the ATA interface. The GRLinux include tested bitfiles for various boards, a precompiled kernel with the appropriate features to run the root file system from a hard drive, and a temporary image for preparing the hard drive with splack. The splack distribution was compiled with support for FPU and integer multiplier.

The kernel needs support for Gaisler ATA controller when accessing the root file system from an ATA disk or compact flash.

10.1 Installing the kernel onto flash

The precompiled kernel is loaded onto the boot PROM of the board, grmon can be used to program the PROM/Flash.

```
$ grmon -jtag
grmon> flash erase all
grmon> flash load image.flashbz
```

The boot loader is the first to be executed, it copies the kernel into main memory and before it starts the execution of the kernel, the kernel command line string is made available. The kernel needs to know what partition on the disk to search for the file system, that can be provided through the kernel command line:

root=/dev/hda1

The example boot line above make the kernel search the first partition on the first disk for a valid file system. In principle the configuration is similar to a NFS root file system as described earlier, instead of telling the kernel the IP address of the server and the location of the share on that server, the kernel is fed with hard disk and partition number.

10.2 Preparing the hard drive

The preparation of the hard drive can be made using a PC computer, or as described below by using the target hardware. Another image besides the flash image has been create for a single purpose, preparing the hard drive from the target hardware using NFS to access the splack distribution. This image is called *image-nomount.dsu*.

Booting the Linux kernel can be done as follows:

```
$ grmon -jtag -nb
grmon> load image-nomount.dsu
grmon> run
```

In order to copy the splack distribution onto the disk one must be able to access it. In this example NFS is used to access the splack distribution:

```
# mount -t nfs 192.168.0.32:/home/daniel /mnt/nfs
```

In order for the kernel to read from the hard disk one should create a partition table and a partition on the hard drive with a file system which the kernel can read the root file system from, EXT2 typically (type 0x83). One can edit the partition table and create new partitions with *fdisk*, it can be started as follows:

```
# fdisk /dev/hda

n   add a new partition
p   print the partition table
o   create a new empty DOS partition table
w   write table to disk and exit
```

When a partition has been created it is possible to format it using the *mke2fs* utility, and copy the splack distribution to the newly created file system:

```
# mke2fs /dev/hda1
# mount -t /dev/hda1 /mnt/hd
# cd /mnt/hd
# tar -zxpf /mnt/nfs/splack-1.0.tar.gz
# cd ..
# umount /mnt/hd
# umount /mnt/nfs
```

10.3 Running splack

If not already connected, connect the hard drive to the target board. Make sure a terminal emulator is connected to the serial port at 38400 baud. The first boot after power up will fail on the lattice board because the DDR RAM needs to be manually reset. Push the reset button and the kernel should boot and mount the CF disk.

The root password is set to "qwerqwer".

One must always terminate the system using the 'shutdown -h' command before switching of the power. This will sync the file systems, otherwise data can get corrupted or lost. Before it is okay to turn power off (or reset it) a message will be printed at the console:

```
Shutdown: hda
Power down.
```

11 INSTALLING DEBIAN 3.1 ON LEON LINUX

Debian is a widely used Linux distribution freely available at www.debian.org. Debian is available platforms, including SPARC. Debian binaries expect FPU and hardware integer multiplier to be available. For further installation information refer to the installation manual at Debian's homepage.

Normally, when installing Debian, installation diskettes or compact discs are used to boot Debian Installation program. However, installing Debian on LEON involves a different approach where it is installed from an UNIX host directly onto a disk or a NFS share. The Installation procedure is split in two stages, first Debian binaries are downloaded and verified from the internet the second stage involves running the binaries on a LEON target board doing the “real” installation and configuration.

In this example Debian is installed onto a NFS share that will later become the root file system of the LEON board. The LEON board is a low-cost GR-XC3S-1500 board from www.gaisler.com.

11.1 Preparing LEON target

Preparing the LEON target to run the installation binaries is done similar to a NFS root file system set up as described previous, the important applications and kernel configuration is listed in two tables below. The LEON system consist of a kernel that is using a NFS share with *BusyBox* as it's root file system.

The BusyBox root file system is exported from a PC as */export/busybox* and the Debian root file system is placed in */export/busybox/debian*.

Name	**Function**
Networking	
NET	Networking support
INET	TCP/IP protocol
IP_PNP	IP settings can be set from kernel command line
Device Drivers / Network device support	
NETDEVICES	Enables the network device driver interface
General machine setup / Grlib: AMBA / Gaisler	
GRLIB_GAISLER_GRETH	Add Gaisler Ethernet 10/100/1000 driver. Select a unique Ethernet address in the MSB and LSB fields.
Device Drivers / Block Devices	
BLK_DEV_INITRD	**Disable** ROMFS RAM root file system
File systems / Network file systems	
NFS_FS	Network file system support
NFS_V3	Support for version 3 of the NFS protocol
ROOT_NFS	Add support for root file system over NFS

Table 11.1: Kernel configuration for Debian and install program

Name	Function
Core Applications	
Bash	Shell to parse Debian scripts
BusyBox	
ar, cat, chmod, chown, chroot, find, grep, gunzip, head, init, ln, md5sum, mkdir, mv, printf, rm, sed, sleep, sort, sync, tar, touch, tr, umount, wc	

Table 11.2: Root file system mounted via NFS

11.2 Installing Debian installation utility to PC and LEON target

The utility that is able to download, extract and install Debian is called *debootstrap*. It can be downloaded from Debian's homepage. Since *debootstrap* is run from the PC as well as the LEON target two versions are needed to be downloaded, the SPARC version for the LEON and the i386 version for the PC. Unless of course running on Solaris.

It is assumed that the Debian package manager is not available. Download and install debootstrap on a non-debian machine as root:

```
$ BUSYBOXLINUX=/export/busybox_rootfs
$ DEBOOT=debootstrap_0.2.45-0.2_i386.deb
$ TMPDIR=/tmp/debinst
$ ARCH=sparc
$ INSTDIR=$BUSYBOXLINUX/debian
$ DISTNAME=sarge
$ MIRROR=ftp://ftp.se.debian.org/debian

$ mkdir -p $TMPDIR/work
$ cd $TMPDIR

$ wget http://ftp.se.debian.org/debian/pool/main/d/debootstrap/$DEBOOT
$ cd work
$ ar -xf $TMPDIR/$DEBOOT

$ cd /
$ tar -zxvf $TMPDIR/work/data.tar.gz
```

Download and install debootstrap to LEON BusyBox root file system:

```
$ DEBOOT=debootstrap_0.2.45-0.2_sparc.deb

$ mkdir -p $TMPDIR/work_sparc
$ cd $TMPDIR

$ wget http://ftp.se.debian.org/debian/pool/main/d/debootstrap/$DEBOOT
$ cd work_sparc
$ ar -xf $TMPDIR/$DEBOOT

$ cd $BUSYBOXLINUX
$ tar -zxvf $TMPDIR/work_sparc/data.tar.gz
```

The two debootstrap applications should now be working both for the LEON target and the PC.

11.3 Downloading Debian binaries using PC

Downloading Debian binaries to PC:

```
$ mkdir -p $INSTDIR
/usr/sbin/debootstrap --download-only --arch $ARCH $DISTNAME $INSTDIR
$MIRROR
```

11.4 Installing Debain binaries from LEON target

The LEON target is able to access the binaries downloaded by the PC in the previous step by entering the */debian* directory. Invoking the Debian installation utility *debootstrap* with the correct parameters makes it continue the installation process, this step may take some time to complete:

```
/usr/sbin/debootstrap sarge /debian
```

11.5 Adding a serial console to Debian

Even though the system console may be the serial terminal no shell is given unless explicitly telling *init* to launch one. This can be done as previously described by editing the */etc/inittab* in the Debian root file system. Add or uncomment:

```
T0:234:respawn:/sbin/getty -L ttyS0 38400 vt100
```

Getty will present us with a login prompt at serial channel 0. One must make sure there is a valid serial terminal device node for *getty* to open in */dev/ttyS0*. If the device is missing it can be created by *mknod*:

```
# cd $INSTDIR/dev
# mknod ttyS0 c 4 64
```

In case of trouble booting Debian it may be of good practice to change the run level to 1 or 2 in *inittab* before booting:

```
# The default runlevel.
id:2:initdefault:
```

11.6 Changing root directory and booting Debian

Before booting the kernel command line needs to be updated to reflect the new root file system. Enter the “make xconfig” GUI and update the kernel command line to include the debian directory:

```
nfsroot=192.168.0.20:/export/busybox_rootfs/debian
```

Make the configuration, rebuild the image and run the image:

```
$ make xconfig
$ make image
$ grmon -jtag -nb
$ load images/image.dsu
$ run
```

11.7 Adding a telnet server to Debian

During the development process it often comes in handy with a *telnet* terminal. The telnet terminal isn't limited by the bandwidth as the serial terminal, making it to an excellent choice when running or debugging new applications.

The easiest way of installing applications in Debian is using the *apt-get* utility. The *apt-get* utility can be setup to fetch binaries from close mirror servers by editing */etc/apt/sources.list*. Setting it up on the LEON target:

```
# echo deb ftp://ftp.se.debian.org/debian stable main contrib non-free >
/etc/apt/sources.list
# apt-get update
```

Installing telnet server can be done as follows:

```
# apt-get install telnetd
```

It is generally an good idea to have a look in the */etc/inetd.conf* and verifying that *telnetd* is correctly configured. To make the telnet server appear on the network *inetd* may need to be restarted to reread it's configuration file, sometimes it is enough sending *inetd* it the SIGHUP signal.

11.8 Installing X.org X11 Server

The graphical X server is normally operated using keyboard and mouse. See the chapter "PS/2 and VGA" on how to setup the monitor and PS/2 keyboard. The peripherals and monitor/GRVGA is setup in */etc/X11/xorg.conf* installed with *apt-get*.

Installing the graphical X server, *fbset* (a frame buffer utility), X fonts and *xterm* terminal emulator is similar to installing the *telnet* server in the previous section:

```
# apt-get install xserver-xorg
# apt-get install xfonts-base
# apt-get install fbset
# apt-get install xterm
```

Configure the X server by editing /etc/X11/xorg.conf.

After setting up the X server properly it is possible to start the server with extra debug output by adding the option -*verbose [level]*:

```
$ X -verbose 3
```

12 SUPPORT

For support, contact the Gaisler Research support team at support@gaisler.com.