Digi ConnectCore User's Guide for Command Line Tools

90000854_B

© Digi International Inc. 2007. All Rights Reserved.

The Digi logo and ConnectCore are trademarks or registered trademarks of Digi International, Inc.

All other trademarks mentioned in this document are the property of their respective owners.

Information in this document is subject to change without notice and does not represent a commitment on the part of Digi International.

Digi provides this document "as is," without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

This product could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes may be incorporated in new editions of the publication.

Digi International Inc.

11001 Bren Road East

Minnetonka, MN 55343 (USA)

☎ +1 877 912-3444 or +1 952 912-3444

http://www.digi.com

Contents

1. Concepts....5
   - 1.1. Embedded Linux Concepts....5
   - 1.2. Digi Embedded Linux concepts....7
   - 1.3. Structure of Digi Embedded Linux....7
   - 1.4. Supported Platforms....8
   - 1.5. Conventions used in this manual....9
   - 1.6. Abbreviations....10
2. Getting started....11
   - 2.1. Connections and cabling....11
   - 2.2. Configure and open a console session....11
   - 2.3. Configure required daemons....13
   - 2.4. Configure the target's network parameters....15
   - 2.5. Working in the target....16
3. Develop a full Embedded Linux project....21
   - 3.1. Overview of Embedded Linux projects....21
   - 3.2. Creating different projects....21
   - 3.3. Identify project parts and contents....24
   - 3.4. Delete projects....24
4. Develop applications....25
   - 4.1. Create an application....25
   - 4.2. Add C and C++ sample applications....26
   - 4.3. Build the project....27
   - 4.4. Run the application....28
5. Configure the Linux kernel....30
   - 5.1. Kernel configuration options....30
   - 5.2. Built-in features and kernel modules....31
   - 5.3. Platform-specific hardware support....32
   - 5.4. Kernel arguments....32
   - 5.5. Kernel modules....33
   - 5.6. Build the kernel and kernel modules....34
   - 5.7. Install the kernel....34
   - 5.8. Load kernel modules....35
   - 5.9. Modify kernel sources....36
6. Customize the root file system....37
   - 6.1. Configure the rootfs....37
   - 6.2. Put files and folders in the rootfs....39
   - 6.3. Build the rootfs....40
   - 6.4. Install the rootfs....40
   - 6.5. Special files....40
   - 6.6. Built-in applications and services....41
   - 6.7. Launch an application after start-up....43
7. Transfer the system to the target....44
   - 7.1. Basic boot loader commands....44
   - 7.2. U-Boot variables....45
   - 7.3. Test the system....46
   - 7.4. Update the Flash memory....47
   - 7.5. Boot from Flash memory....51

8. **Devices and Interfaces** 52
8.1. Table of devices and their hardware resources 52
8.2. GPIO pins and custom driver 53
8.3. Ethernet interface 59
8.4. Wireless network interface 60
8.5. Flash memory device 62
8.6. Serial device driver 64
8.7. Serial Peripheral Interface (SPI) mode 66
8.8. Touch screen 67
8.9. USB host interface 70
8.10. I^2C 72
8.11. Real Time Clock (RTC) 75
8.12. Video/Graphics support 76
8.13. High-performance counter 78
9. **Use the WLAN adapter** 80
9.1. Wireless security concepts 80
9.2. Features of the WLAN adapter 80
9.3. Include the wireless interface in the Linux kernel 81
9.4. Wireless interface LEDs 82
9.5. Network settings of WLAN interface 83
9.6. Basic wireless operations 83
9.7. Authentication and encryption 85
9.8. Wireless connection examples 86
9.9. Save the configuration 90
9.10. Fine-tune wireless connections 90
10. **Boot loader development** 91
10.1. U-Boot projects 91
10.2. Configure U-Boot 91
10.3. Platform-specific source code 92
10.4. Customize U-Boot 93
10.5. Compile U-Boot 94
10.6. Install U-Boot 94
10.7. Update U-Boot 95
11. **Graphics libraries** 96
11.1. Qtopia core 96
11.2. Qtopia core example applications 96
12. **Troubleshooting** 98
12.1. Getting NFS service when booting with the LiveDVD 98
12.2. Characters returned by target when target is powered off 99
12.3. Writing large files to Flash from U-Boot 99
13. **Recover a device** 100
13.1. JTAG Booster and software 100
14. **Uninstall Digi Embedded Linux** 101
15. **References** 102

1. Concepts

Developing applications for embedded systems differs from developing them for a desktop computer. Embedded-system applications involve several more elements than the applications themselves, such as the operating system and any necessary customization of it, the hardware drivers, the file system, and other elements. This topic introduces the software elements of an embedded system and the development environment needed to create them.

1.1. Embedded Linux Concepts

Embedded systems are ubiquitous. These dedicated small computers are present in communications systems, transportation, manufacturing, detection systems, and many machines that make our lives easier.

The open nature of Linux and its availability for many different hardware architectures makes it the perfect candidate for embedded platforms.

1.1.1. Cross-compilation

Whenever code is generated for an embedded target on a development system with a different microprocessor architecture a cross-development environment is needed. A cross-development compiler is one that executes in the development system, for example, an x86 PC, but generates code that executes in a different processor, for example, if the target is ARM.

Digi Embedded Linux provides the GNU cross-development toolchain for ARM architectures, which contains the compiler, linker, assembler, and shared libraries needed to generate software for the supported platforms.

1.1.2. Boot loader

A boot loader is a small piece of software that executes soon after powering up a computer. On a desktop PC, the boot loader resides on the master boot record (MBR) of the hard drive, and is executed after the PC BIOS performs various system initializations. The boot loader then passes system information to the kernel, for instance, the hard drive partition to mount as root, and finally executes the kernel.

In an embedded system, the role of the boot loader is more complicated, since these systems do not have a BIOS to perform the initial system configuration. The low-level initialization of the microprocessor, memory controllers, and other board-specific hardware varies from board to board and CPU to CPU. These initializations must be performed before a Linux kernel image can execute.

At a minimum, a boot loader for an embedded system performs the following functions:

- Initializes the hardware, especially the memory controller.
- Provides boot parameters for the operating system image.
- Starts the operating system image.

Additionally, most boot loaders also provide convenient features that simplify development and update of the firmware, such as:

- Reading and writing arbitrary memory locations.
- Uploading new binary images to the board's RAM via a serial line or Ethernet
- Copying binary images from RAM to Flash memory.

1.1.3. Kernel

The kernel is the fundamental part of an operating system. It is responsible for managing the resources and the communication between hardware and software components.

The kernel offers hardware abstraction to the applications and provides a secure access to the system memory. It also includes an interrupt handler that handles all requests or completed I/O operations.

1.1.3.1. Kernel modules

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.

For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, Linux developers would have to build monolithic kernels and add new functionality directly into the kernel image. Besides resulting in larger kernels, working without kernel modules would present the disadvantage of requiring rebuilding and rebooting the kernel every time new functionality was added.

In embedded systems, where functionality can be activated depending on the needs, kernel modules become a very interesting way of adding features without enlarging the kernel image size.

1.1.4. Root file system

Operating Systems normally rely on a set of files and directories. The root file system is the top of the hierarchical file tree. It contains the files and directories critical for system operation, including the device directory and programs for booting the system. The root file system also contains mount points where other file systems can be mounted to connect to the root file system hierarchy.

1.1.5. Applications

Software applications are programs that employ the capabilities and resources of a computer to do a particular task.

Applications make use of hardware devices by communicating with device drivers, which are part of the kernel.

1.2. Digi Embedded Linux concepts

1.2.1. Projects

The philosophy of work in Digi Embedded Linux is linked to the idea of 'projects'. A project is actually a folder which contains all the software components required to build a complete solution for the target platform, such as a kernel, a rootfs, a boot loader and user applications.

With just some simple commands, the compilation process takes care of compiling the kernel, the applications, generating the target's file system, and compressing into the final binary images. The compilation process take place within the project folder with normal user permissions.

1.2.2. Workspace

Although projects can be stored anywhere, it is recommended that all projects are stored within a single directory: the workspace. This is just an ordering recommendation.

1.2.3. Makefiles and dependencies

Makefiles are special format files that instruct the utility **make** how to build and manage software projects. The **make** program then helps to develop large software projects by keeping track of which parts of the entire program have changed, building only the parts that have changed since the last build.

The Makefile structure consists of a set of rules and dependencies that define how the project is built. To see a complete description of the **make** utility and Makefiles take a look at the GNU *make* manual at *http://www.gnu.org/software/make/manual/make.html*.

In Digi Embedded Linux Makefile templates are used to generate the main Makefile for the project. That project Makefile has rules to build the complete project with all of its components.

1.3. Structure of Digi Embedded Linux

Digi Embedded Linux allows easy development of software under Linux 2.6 for Digi International Inc. embedded modules. The package contains the complete source code of Linux kernel, and tools to customize rootfs and boot loader and to build user applications. It provides a complete package for configuring, creating, and modifying custom kernels, boot loaders, rootfs and applications for specific embedded systems.

1.3.1. Digi EL directory tree

Digi Embedded Linux package has the following directory tree:

```
$   cd /usr/local/DigiEL-4.0
$   ls
apps               bootloader  images     libexec        scripts
arm-linux          configs     include    mkproject.sh   templates
arm-linux-uclibc   digiesp     kernel     modules        uninstall
bin                docs        lib        rootfs
$
```

Following is a brief description of all the directories and files of the Digi Embedded Linux package. Most of them are part of the toolchain. The structure of the toolchain directories is fixed:

- **apps**: User applications templates.
- **arm-linux**: Part of the toolchain; symlink to **arm-linux-uclibc**.
- **arm-linux-uclibc**: Part of the toolchain; several binary tools.
- **bin**: Part of the toolchain; cross-compilers, linkers. There is also some extra-toolchain tools included that are useful for different stages of project building, for example tools to create rootfs images of different types (cramfs, jffs2).
- **bootloader**: Source code of U-Boot boot loader.
- **configs**: Default config files for our target platforms.
- **digiesp**: Digi Embedded Linux Eclipse plugin.
- **docs**: Documentation for Digi Embedded Linux.
- **images**: Images ready to be programmed in a target device.
- **include**: Part of the toolchain; header files required to build applications.
- **kernel**: Linux kernel source code.
- **lib**: Part of the toolchain; cross-compiled libraries.
- **libexec**: Part of the toolchain.
- **mkproject.sh**: Project creation script. Run it without options to see its associated help.
- **modules**: External kernel modules templates.
- **rootfs**: A base rootfs for all the projects and the structure to add prebuilt applications at project configuration time (**rootfs_extras**).
- **scripts**: Several shell scripts used for different tasks, such as make images, check libraries, parse headers, etc.
- **templates**: Makefile templates for different types of projects. These are used to generate the project main Makefile.
- **uninstall**: Uninstalls the Digi Embedded Linux binary.

1.4. Supported Platforms

This document applies to and mentions to the following supported platforms:

- Digi ConnectCore 9C platform
- Digi ConnectCore Wi-9C platform
- Digi ConnectCore 9P platform

Where instructions use the keyword ***platformname***, substitute ***platformname*** with the actual platform being used:

If using this module:	Specify this platform name
Digi ConnectCore 9C	cc9cjsnand
Digi ConnectCore Wi-9C	ccw9cjsnand
Digi ConnectCore 9P	cc9p9360js

Depending the platform being used, the information in dialogs and output messages may vary from that shown this manual.

1.5. Conventions used in this manual

This manual uses the following typographical conventions to display and refer to information:

Convention	Use
Style	New terms and variables in commands, code, and other input.
`Style`	In examples, to show the contents of files, the output from commands. In, text the C code.
	Variables to be replaced with actual values are shown in italics.
`Style`	Items in the Digi ESP interface, such as menu items, dialogs, tabs, buttons, and other controls.
	In examples, to show the text that should be typed literally by the user.
#	A prompt that indicates the action is performed in the target device.
$	A prompt that indicates the action is performed in the host computer.

It also uses these frames and symbols:

This is a warning. It helps solve or to avoid common mistakes or problems

This is a hint. It contains useful information about a topic

```
$   This is a host computer session
$   Bold text indicates what must be input
```

```
#   This is a target session
#   Bold text indicates what must be input
```

1.6. Abbreviations

ASCII	American Standard Code for Information Interchange
BIOS	Basic Input Output System
CPU	Central Processing Unit
CVS	Concurrent Versions System
DAC	Digital to Analog Converter
DHCP	Dynamic Host Configuration Protocol
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GDB	GNU Debugger
GNU	GNU's Not UNIX
GPIO	General Purpose Input/Output
HID	Human Interface Device
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IOCTL	I/O Control
IP	Internet Protocol
IRQ	Interrupt Request
JFFS2	Journaling Flash File System version 2
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
MBR	Master Boot Record
MTD	Memory Technology Device
NFS	Network File System
NVRAM	Non-volatile RAM
OS	Operating System
PC	Personal Computer
PID	Process Identification
RAM	Random Access Memory
ROOTFS	Root File System
RTC	Real-Time Clock
SPI	Serial Peripheral Interface
TFT	Thin Film Transistor
TFTP	Trivial File Transfer Protocol
UID	User Identification
USB	Universal Serial Bus
VGA	Video Graphics Array
VM	Virtual memory

WLAN	Wireless Local Area Network

2. Getting started

This topic explains how to connect the development board (target) and configure a host PC to connect to it.

2.1. Connections and cabling

Connect the hardware as explained in the *Quick Start Guide*.

2.2. Configure and open a console session

The target board prints out messages on the serial port. To be able to see these messages, it is necessary to start a console session with the target by means of a Linux communications program, like **minicom** or **seyon**.

The default serial communication parameters are 38400 baud, no parity, 8 data bits, and 1 stop bit. In Linux, the serial ports device nodes are normally at **/dev/ttySn** (where **n** is the number of port).

*Unless otherwise stated, this guide assumes the target is connected to the first serial port (COM 1, /dev/ttyS0) of the host. If using another port, change **n** in **ttySn** to the appropriate number.*

In some Linux distributions, the serial ports have restricted access. If the serial port cannot be opened, consult the Linux administrator.

2.2.1. Minicom

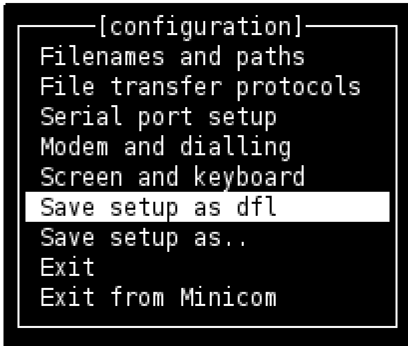
Minicom is the more popular Linux communications program. Before launching **minicom**, configure it. To do so, login in as root and issue the command:

```
#   minicom -s
```

Then go to **Serial port setup** and change the values to as needed:

```
A -    Serial Device      : /dev/ttyS0
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 38400 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

   Change which setting?
```

When all parameters are set, select **Save setup as dfl** to save the configuration.



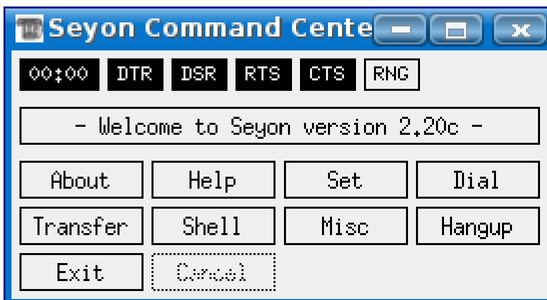
Next time start **minicom** as a standard user with:

```
$   minicom
```

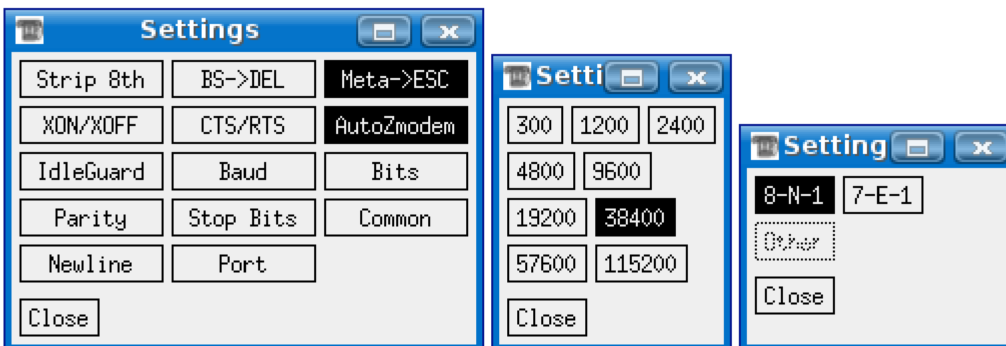
2.2.2. Seyon

Seyon is a complete full-featured telecommunications package for the X Window System. To use it start **seyon** as a standard user by entering:

```
$   seyon -modems /dev/ttyS0
```



In the **Seyon Command** window, press the **Set** button to open the settings window. In the settings window, the communication parameters can be adjusted.



Press the **Baud** button, and select 38400. Press the **common** button, and select 8-N-1.

2.3. Configure required daemons

This topic shows how to configure additional services that are required to prepare the development computer to work with the target. If Digi Embedded Linux was installed together with the provided Kubuntu Linux distribution, a TFTP server and an NFS server are already installed and configured, and this topic can be skipped.

2.3.1. TFTP daemon

The U-Boot boot loader running in the target board is able to write files to the Flash memory of the module. A TFTP server is required to transport these files from the host computer to the target.

Debian-like distributions users can execute the following command to install a TFTP server:

```
#   apt-get install tftpd
```

After completing installation, create a directory **/tftpboot** where exported files will be located. (Must be root user to create this directory.) Images can be placed in the directory automatically by Digi Embedded Linux build environment. Activate all the permissions of this folder.

```
#   mkdir /tftpboot
#   chmod 1777 /tftpboot
```

To make sure the TFTP server is using the **/tftpboot** directory, check the Internet daemons configuration file **/etc/inetd.conf**. It should contain an entry similar to the following:

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd -s /tftpboot
```

This line is usually added as part of the process of the daemon installation in Debian-like distributions. If the entry is not there, use an editor and change the file accordingly. For specific options for tftpd, see the Linux man page for the command:

```
$   man tftpd
```

2.3.2. NFS server

The network file system (NFS) simplifies application development on the target. NFS allows a target board to mount over Ethernet a host computer directory as its root file system with read/write permissions. NFS also allows access to the file system from the target and host computer at the same time.

NFS server configuration details are very specific to the various distributions and beyond the scope of this help. This help describes the necessary modifications on hosts running a Debian-like distribution only. To set up an NFS server using a different distribution, see the Linux distribution manual.

When the NFS server package (package nfs-kernel-server) is installed on a Debian-like distribution, the file **/etc/exports** contains information on exported directories and their access rights. For detailed information about the **/etc/exports** file, refer to the Linux man pages.

Add the following line to the **/etc/exports** file to provide read/write access for the target:

```
BOOTDIR    IP_ADDRESS(rw,all_squash,anonuid=YOUR_UID,anongid=YOUR_GID,async)
```

Replace **BOOTDIR** with the path to the NFS directory which is exported to the target, **IP_ADDRESS** with the IP address of the target, **YOUR_UID** with your user UID and **YOUR_GID** with your user GID.

Use the commands **id -u** and **id -g** to obtain your user UID and GID:

```
$   id -u
1000
$   id -g
1000
```

By default, the build process copies the target's rootfs to **/export/nfsroot-*platformname*** (see topic 1.4), For example, to export the rootfs for a ConnectCore Wi-9C platform to a target with IP address 192.168.42.30, write the following to **/etc/exports**:

```
/exports/nfsroot-ccw9cjsnand
192.168.42.30(rw,all_squash,anonuid=1000,anongid=1000,async)
```

Or, for simplicity's sake, export the whole **/exports** directory for a complete subnet:

```
/exports 192.168.42.0/24(rw,all_squash,anonuid=1000,anongid=1000,async)
```

After modifying the **/etc/exports** file, restart the NFS server with the following command (as root):

```
#   /etc/init.d/nfs-kernel-server restart
```

2.4. Configure the target's network parameters

Because communication between the Digi ESP environment and target occurs over Ethernet, network settings must be configured on the target. This is done by changing some variables of the boot loader. The process is explained in detail in topic 7.

Power on the development board with the power switch. The LEDs on the board light up, and 2 seconds later, the system prints boot messages on the Serial Console view. To stop the autoboot process, with the focus on the Serial Console view, press any key.

To configure the network settings of the target (IP, mask, the IP address of the host, etc.), enter the following commands in the Serial Console view:

```
#   setenv ipaddr XXX.XXX.XXX.XXX
#   setenv netmask NNN.NNN.NNN.NNN
#   setenv ipaddr_wlan WWW.WWW.WWW.WWW
#   setenv netmask_wlan MMM.MMM.MMM.MMM
#   setenv serverip YYY.YYY.YYY.YYY
#   saveenv
```

where:

- ***XXX.XXX.XXX.XXX*** is the IP address for the target's Ethernet interface.
- ***NNN.NNN.NNN.NNN*** is the target's Ethernet network mask.
- ***WWW.WWW.WWW.WWW*** is the IP address for the target's WLAN adapter (only if the module is a ConnectCore Wi-9C).
- ***MMM.MMM.MMM.MMM*** is the wireless network mask (only if the module is a ConnectCore Wi-9C).
- ***YYY.YYY.YYY.YYY*** is the IP address of the development workstation.

The Ethernet IP addresses of the target and the host PC must be in the same network segment. For a ConnectCore Wi-9C module, the Wireless IP addresses of the target and the AP must also be in the same network segment.

The **saveenv** command saves the target's network settings in NVRAM. As a final step, switch off the target again.

2.5. Working in the target

Now that everything is properly set up, the next step is to work with the target. Power on the development board with the power switch. After power-on, the LEDs on the board will light up, and 2 seconds later, the system will print boot messages on the Serial Console view. To let the target boot automatically, do not press any key. After 25-30 seconds, the boot loader unpacks and launches the pre-installed Linux kernel from the built-in Flash memory.

During this process, output messages on the terminal client similar to the output below are displayed.

```
U-Boot 1.1.4 (Feb 20 2007 - 14:23:03) DEL_4_0_RC3
for Digi ConnectCore Wi-9C on Development Board

DRAM:  64 MB
NAND:  128 MiB
In:    serial
Out:   serial
Err:   serial
CPU:   NS9360 @ 154.828800MHz
Strap: 0x03
SPI ID:2007/01/25, V1_4rc2, CC9C/CCW9C, SDRAM 64MByte, CL2, 7.8us, LE
FPGA:  wifi.ncd, 2007/01/25, 17:49:41, V2.01
Hit any key to stop autoboot:  0
...
...
[LINUX KERNEL BOOT MESSAGES]
...
...
Starting dropbear sshd: OK
Starting ftp server: vsftpd.
Starting boa webserver: boa.


BusyBox v1.2.2 (2007.01.16-12:10+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ #
```

2.5.1. Common Linux commands (busybox)

After Linux starts successfully, two things are available: a root file system, and a shell, as part of the executable file **busybox**.

busybox is an executable file that contains small versions of many common UNIX tools. These smaller versions serve as replacements for most utilities found in desktop Linux distributions, with the advantage of being small enough to be useful for embedded systems.

busybox includes the most-used shell applications, such as **cat**, **chmod**, **echo**, **mount**, etc. These utilities generally have fewer options than their full-featured desktop's equivalents; however, the included options provide the expected functionality. Further, in Digi Embedded Linux, less-important shell applications have been stripped off in order to obtain a small *busybox* binary.

To list currently included applications in **busybox**, on the Serial Console, type **busybox** without arguments.

```
#   busybox
BusyBox v1.2.2 (2006.12.18-19:46+0000) multi-call binary

Usage: busybox [function] [arguments]...
   or: [function] [arguments]...

       BusyBox is a multi-call binary that combines many common Unix
       utilities into a single executable. Most people will create a
       link to busybox for each function they wish to use and BusyBox
       will act like whatever it was invoked as!

Currently defined functions:
       [, [[, addgroup, adduser, ash, awk, basename, bbconfig, bunzip2,
       busybox, bzcat, cal, cat, chgrp, chmod, chown, chroot, chvt, clear,
       cp, cut, date, dd, deallocvt, delgroup, deluser, df, diff, dirname,
       dmesg, dos2unix, du, dumpleases, echo, egrep, env, expr, false,
       fbset, fdisk, fgrep, find, free, ftpget, ftpput, fuser, getopt,
       getty, grep, gunzip, gzip, halt, head, hexdump, hostid, hostname,
       hwclock, id, ifconfig, init, insmod, install, kill, killall, klogd,
       less, ln, logger, login, logname, logread, losetup, ls, lsmod,
       makedevs, md5sum, mdev, mkdir, mknod, mkswap, mktemp, modprobe,
       more, mount, mountpoint, mv, nc, netstat, nice, nslookup, od,
       openvt, passwd, pidof, ping, pivot_root, poweroff, printenv, ps,
       pwd, rdate, readlink, reboot, renice, reset, rm, rmdir, rmmod,
       route, run-parts, sed, seq, setsid, sh, sha1sum, sleep, sort,
       start-stop-daemon, stat, strings, stty, su, sulogin, swapoff,
       swapon, sync, sysctl, syslogd, tail, tar, tee, telnet, telnetd,
       test, tftp, time, top, touch, tr, true, tty, udhcpc, udhcpd, umount,
       uname, uniq, unix2dos, unzip, uptime, usleep, uudecode, uuencode,
       vi, vlock, watch, wc, wget, which, who, whoami, xargs, yes, zcat

#
```

Brief help text is provided for each command. To display this help, enter:

```
#   command_name --help
```

For example, to display help about the **copy** (**cp**) command, enter:

```
#   cp --help
BusyBox v1.2.2 (2006.12.18-19:46+0000) multi-call binary

Usage: cp [OPTION]... SOURCE DEST
Copies SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

    -a,         Same as -dpR
    -d,-P       Preserves links
    -H,-L       Dereference all symlinks (implied by default)
    -p          Preserves file attributes if possible
    -f          force (implied; ignored) - always set
    -i          interactive, prompt before overwrite
    -R,-r       Copies directories recursively

#   cat --help
BusyBox v1.2.2 (2006.12.18-19:46+0000) multi-call binary

Usage: cat [-u] [FILE]...

Concatenates FILE(s) and prints them to stdout.

Options:
    -u          ignored since unbuffered i/o is always used

#
```

To list files, use the **ls** command. To navigate through the directories, use the **cd** command.

To become familiar with using the **busybox** shell, try other commands.

2.5.2. Open a Telnet session

A Telnet server (telnetd) is included and started by default, so a Telnet session can be opened from the host computer.

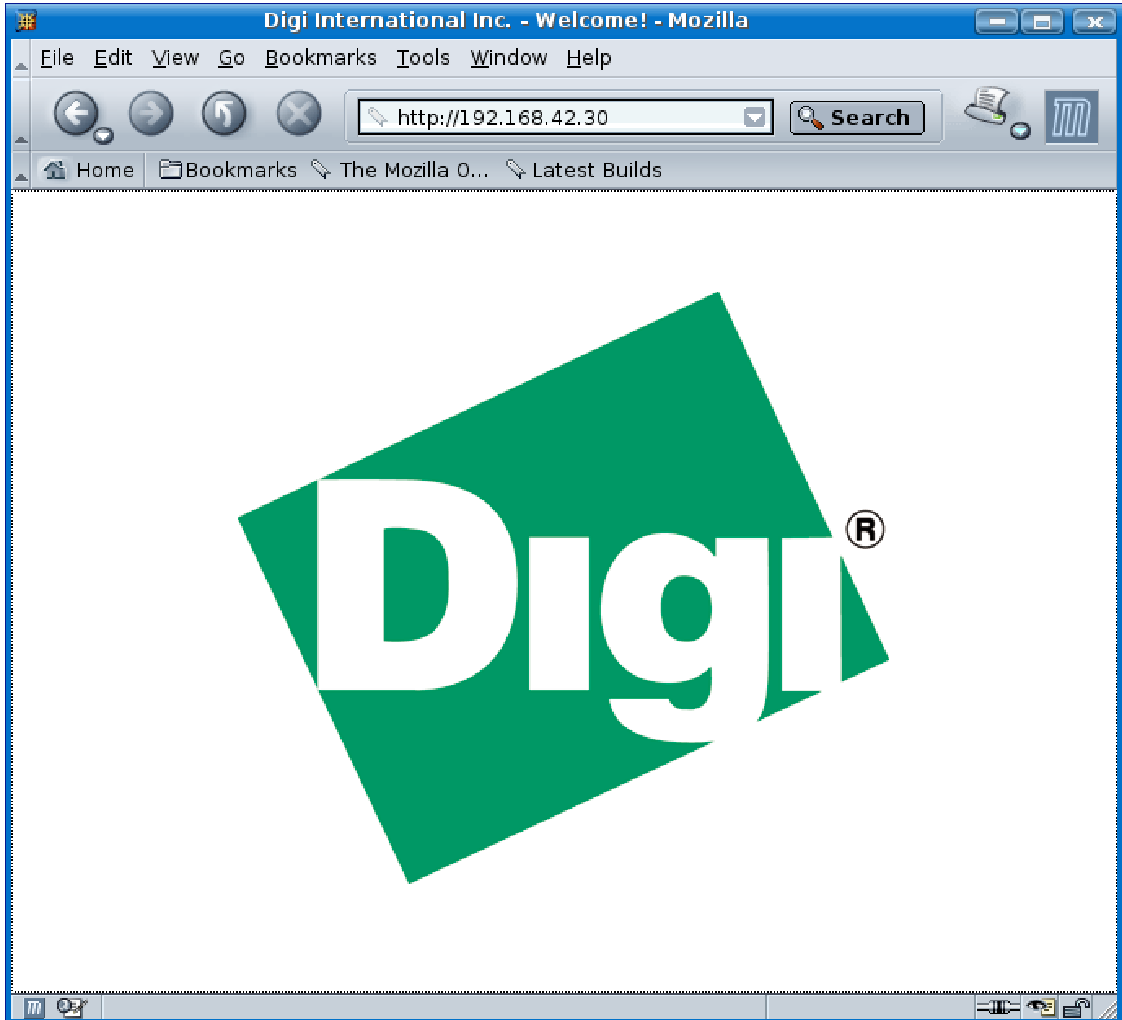
```
$   telnet 192.168.42.30
Trying 192.168.42.30...
Connected to 192.168.42.30.
Escape character is '^]'.



BusyBox v1.2.2 (2006.12.18-19:46+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ #
```

This will open a **busybox** shell in the Telnet session. There is no need to provide a username and password to log in the target.

2.5.3. Connect to the web server

A webserver, called **boa webserver**, is included as an extra package. This is a small single-tasking HTTP server. Connecting from the host with any web browser displays a simple web page with the Digi logo. The configuration file for the webserver is named **boa.conf** and is located in the ***/etc/boa/*** directory on the target.

Open a browser and type the IP address of the target. The default IP address is 192.168.42.30.



2.5.4. File transfer to the target (FTP)

An FTP server (vsftpd) is included as an extra package. FTP sessions can be opened from the host computer to the target board. The FTP server can be connected to as user **anonymous** or **ftp** without password.

The FTP server allows to transfer files between host and target. To upload a file:

1. Change to the folder in the host where the file is.
2. From the folder where the file is, open an FTP connection to the target.
3. Change the target's current directory to /tmp, which has write access.
4. Upload the file with the FTP command “put <file>”.
5. Check that the file has been transferred doing an 'ls' command.
6. Close the FTP connection with “exit”.

```
$   cd /usr/local/DigiEL-4.0/
$   ftp 192.168.42.30
Connected to 192.168.42.30.
220 (vsFTPd 2.0.5)
Name (192.168.42.1:myuser): ftp
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd tmp
250 Directory successfully changed.
ftp> put mkproject.sh
local: mkproject.sh remote: mkproject.sh
200 PORT command successful. Consider using PASV.
150 Ok to send data.
226 File receive OK.
13211 bytes sent in 0.01 secs (888.6 kB/s)
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rwxrwxr--    1 1001     1001        13211 Jan 03 09:18 mkproject.sh
226 Directory send OK.
ftp> exit
221 Goodbye
```

3. Develop a full Embedded Linux project

This topic covers the creation of Digi Embedded Linux projects with its different components. It is possible to create separate projects for each part (boot loader, kernel, rootfs, applications), or a full project with all the components integrated.

This topic involves creating a project which will be used and modified later.

3.1. Overview of Embedded Linux projects

An embedded Linux *project* is a set of software components that are part of the same solution for a target platform.

As described in topic 1.1, the different parts of an embedded system are the boot loader, the kernel, the rootfs, and the applications. Since these parts are independent of each other, it is possible to create a kernel-only project or an applications-only project, for example.

Most of the time, however, several parts (or every single part) of the embedded software need to be customized. Projects can be created that integrate some or all of the customized parts.

Projects must be given a name, and their files reside on the same directory of the selected workspace.

3.2. Creating different projects

3.2.1. Project wizard script

The project wizard script is a tool that creates the specified projects for the specified platform. The script is called **mkproject.sh** and its options are:

```
$   /usr/local/DigiEL-4.0/mkproject.sh
Usage: mkproject.sh [OPTIONS]

    -a, --with-apps:        create user applications project
    -k, --with-kernel:      create kernel project
    -l, --list-platforms:   list the available platforms in the environment
    -m, --with-modules:     create kernel modules (depends on --with-kernel)
    -r, --with-rootfs:              create rootfs project
    -u, --with-uboot:               create U-Boot project
    --enable-platform=<platform>:   select platform for the project
    --tftp-dir=<dir>:               select tftp directory
    --nfs-dir=<dir>:                select nfs  directory
The available platforms are:

       cc9cjsnand   cc9p9360js   ccw9cjsnand
```

3.2.2. Kernel project

Consider a scenario where different equipment is being designed, with different functionality, but with the same kernel requirements, in terms of network services or file-systems support. The targets will have different applications running on them and different root file systems, but they can have the same kernel. For this kind of scenario, a *kernel-only project* can be created, which consists of developing and configuring a kernel to run on all the targets.

To create a kernel-only project (with kernel modules) for a ConnectCore Wi-9C, create the kernel project folder and execute the project wizard script from there with only kernel and kernel modules support:

```
$   mkdir myKernelProject
$   cd myKernelProject
$   /usr/local/DigiEL-4.0/mkproject.sh -km --enable-platform=ccw9cjsnand
    --tftp-dir=/tftpboot --nfs-dir=/exports/nfsroot-ccw9cjsnand
```

*Projects can be created for different platforms by changing the value of the **–enable-platform** argument, as seen in topic 1.4.*

Now the kernel project is ready. Configuring, building, and installing the kernel is explained in topic 5.

3.2.3. Rootfs project

Suppose a target has a running kernel but requires a different root file system in it, including a web page or different applications. To do so, a rootfs-only project is created to generate a new root file system for the target device.

To create a Digi Embedded Linux rootfs-only project for a ConnectCore Wi-9C, create the rootfs project folder and execute the project wizard script from there with only rootfs support:

```
$   mkdir myRootfsProject
$   cd myRootfsProject
$   /usr/local/DigiEL-4.0/mkproject.sh -r --enable-platform=ccw9cjsnand
    --tftp-dir=/tftpboot --nfs-dir=/exports/nfsroot-ccw9cjsnand
```

*Projects can be created for different platforms by changing the value of the **–enable-platform** argument, as seen in topic 1.4.*

Now the rootfs project is ready. Configuring, building, and installing the rootfs is explained in topic 6.

3.2.4. Applications project

To create one or more applications that can be run on a certain target, create an applications-only project. These applications can be transferred to the target via FTP, or be put in a storage media (like a Compact Flash card).

To create a Digi Embedded Linux applications-only project for a ConnectCore Wi-9C, create the applications project folder and execute the project wizard script from there with only applications support:

```
$   mkdir myAppsProject
$   cd myAppsProject
$   /usr/local/DigiEL-4.0/mkproject.sh -a --enable-platform=ccw9cjsnand
    --tftp-dir=/tftpboot --nfs-dir=/exports/nfsroot-ccw9cjsnand
```

Projects can be created for different platforms by changing the value of the – ***enable-platform*** *argument,, as seen in topic 1.4.*

Now the applications project is ready. Topic 4 describes developing applications.

3.2.5. U-Boot project

To customize the boot loader of a device or a set of devices (for example, to make it smaller or use custom commands), create a Digi Embedded Linux U-Boot-only project.

To create a U-Boot-only project for a ConnectCore Wi-9C, create the U-Boot project folder and execute the project wizard script from there with only U-Boot support:

```
$   mkdir myUbootProject
$   cd myUbootProject
$   /usr/local/DigiEL-4.0/mkproject.sh -u --enable-platform=ccw9cjsnand
    --tftp-dir=/tftpboot --nfs-dir=/exports/nfsroot-ccw9cjsnand
```

Projects can be created for different platforms by changing the value of the – ***enable-platform*** *argument, as seen in topic 1.4.*

Now the U-Boot project is ready. Topic 10 describes boot loader development.

U-Boot projects copy the complete U-Boot sources (around 55 Mb) to the project directory.

3.2.6. Full Embedded Linux project

If the intention is to develop and customize each single part of a project, components can be integrated into a full-featured project, in a manner similar to selecting separate components for the project.

To create a full-featured Embedded Linux project add the options seen in the previous topics for the different components.

For example, create a full Embedded Linux project to work with it in the following topics. Call this project **myFullProject**. The steps are:

1. Create the project folder
2. Execute the project wizard script from there with Kernel, kernel modules, rootfs, applications and U-Boot support:

```
$   mkdir myFullProject
$   cd myFullProject
$   /usr/local/DigiEL-4.0/mkproject.sh -kmrau
    --enable-platform=ccw9cjsnand --tftp-dir=/tftpboot
    --nfs-dir=/exports/nfsroot-ccw9cjsnand
```

This task consists of adding the options for all types of subprojects.

Although a full Embedded Linux project is being created for demonstration purposes, to keep projects as small as possible, it is recommended that projects be created with the required components only.

3.3. Identify project parts and contents

Here is the structure of the full project created in the previous topic:

```
$   cd <path_to_myFullProject>
$   tree --dirsfirst -L 2
    .
    |-- build
    |   |-- U-Boot
    |   |-- apps
    |   |-- kernel
    |   |-- modules
    |   |-- rootfs
    |   |-- scripts
    |   `-- Makefile
    |-- configs
    |   |-- Kconfig
    |   `-- add_files.sh
    |-- images
    `-- Makefile

    9 directories, 4 files
```

The project folder has three main directories and one Makefile:

- **build**: Stores the files of the build process for the current project. It contains several subfolders and one Makefile.
    - **U-Boot**: this folder stores all the U-Boot code. Each time U-Boot is selected as part of a project all the U-Boot code is copied to this folder.
    - **apps**: this folder stores the user applications. Each user application is in a subfolder of its own below this one.
    - **kernel**: this folder stores the object files (not sources) of the kernel build process.
    - **modules**: this folder stores the user kernel modules. This is the place to create custom kernel modules. Each module has to be in a subfolder of this one.
    - **rootfs**: this folder stores the built rootfs, and is the source for the rootfs which is transferred to the target.
    - **scripts**: It contains the object files of the kconfig/kbuild configuration tool used for configuring the different options for the projects.
    - **Makefile**: this is a kbuild/kconfig configuration tool required file.
- **configs**: Stores the Kconfig file for the current project. This file is dynamically created, depending on the options selected at project creation time, and is the entry point for the kbuild/kconfig configuration tool. It also contains **add_files.sh**, a template script for tweaking the rootfs for a project; for example, creating custom files or folders.
- **images**: Stores the resulting images of the build process (kernel, rootfs, U-Boot).
- **Makefile**: The main Makefile of the project. It is also dynamically created at project creation time depending on the options selected, and it has rules to build the whole project or just the parts specified by the user, for example u-boot, rootfs, applications, modules.

3.4. Delete projects

All files related to a project are stored under the project directory, therefore, to delete a project, delete the project folder.

4. Develop applications

This topic covers developing user applications. Digi Embedded Linux provides sample applications that can be used as templates. Users can also create their own applications starting from scratch. This topic uses the full project created in topic 3.2.6. In the following lines, instructions are given on the assumption that the current working directory is the **myFullProject** directory.

4.1. Create an application

As seen previously in topic 3.3 the subfolder **build/apps/** contains the user applications. Initially, that folder is empty because no user application has been created yet.

To create a new application:

1. Create a subfolder of **build/apps/** with the name of the application.
2. Create the application source code files inside that folder, and provide a Makefile that builds and installs the application. The Makefile is the interface between the project. Specifying a Makefile is required.

As an example, create the famous **hello_world.c** application. First create the application folder:

```
$   mkdir -p build/apps/hello_world
```

Then provide the source code **hello_world.c**:

```
#include <stdio.h>

int main( void )
{
    printf( "Hello World!\n");
    return 0;
}
```

And provide the Makefile (with clean and installation information):

```
ROOTFS_DIR = $(strip $(wildcard $(DEL_PROJ_DIR)/build/rootfs))
STRIP      = $(CROSS_COMPILE)strip
CC         = $(CROSS_COMPILE)gcc
CFLAGS     = -Wall
BINARY     = hello_world

all: $(BINARY)

install: $(BINARY)
ifneq ($(ROOTFS_DIR),)
    $(STRIP) $<
    install -D -m 0755 $< $(ROOTFS_DIR)/usr/bin/$<
else
    $(info *** Directory $(ROOTFS_DIR) not found, $< not installed.)
endif

clean:
    rm -f $(BINARY)
```

As this Makefile is automatically called by the project topdir Makefile, some exported variables are available. This is useful, for example, in the install rule, which uses the variable **DEL_PROJ_DIR** (path to project directory). Using the provided variables is optional. The application can be installed anywhere.

Now the application is ready to be built.

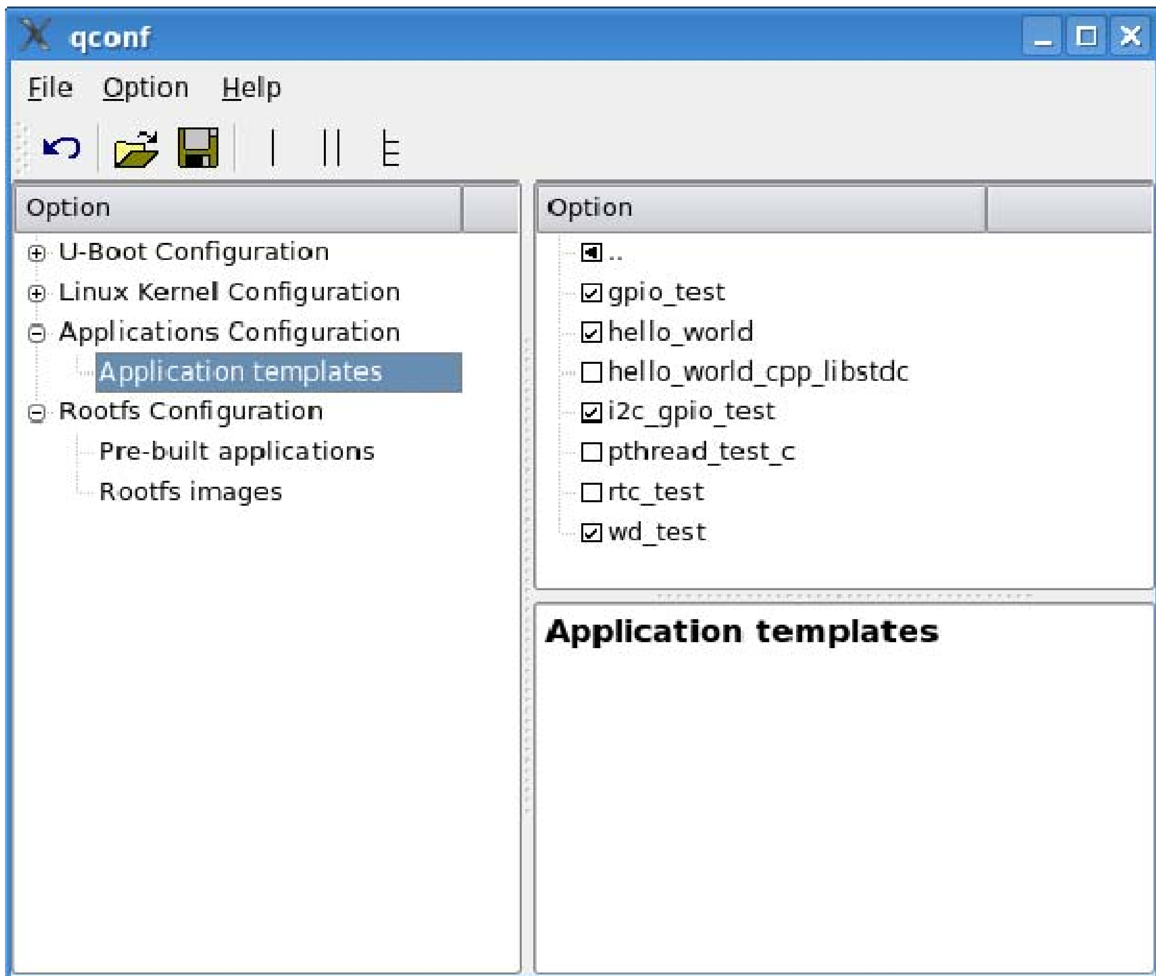
4.2. Add C and C++ sample applications

Digi Embedded Linux provides some example applications that can be used as templates.

To include the application templates in the project run the configuration tool from the project directory:

```
$   make xconfig
```

This window is displayed:



Under **Applications Configuration > User application templates**, select which templates to include in the project. For example, check **gpio_test** and **hello_world**, save, and exit the configuration tool.

The selected templates will be transferred into **build/apps** folder. There, their source files and Makefiles can be modified to meet requirements, or used as help files for developing new applications.

Review the included Makefiles to determine whether any topdir variables can be used in custom Makefiles.

4.3. Build the project

After creating an application from scratch and adding an example application to the project, it is time to build the applications.

From the project folder, execute:

```
$   make build_apps
```

The full project Makefile will go down the **build/apps/** folder and execute the **make** rule for each application subfolder.

Because the application's Makefile also provides an **install** rule the application can be installed by typing:

```
$   make install_apps
```

The full project Makefile will go down the **build/apps/** folder and run the **make install** rule for each application subfolder.

Now the application is built and installed according to each application's Makefile rules.

*Doing a **make** without any arguments builds the complete project, including applications, kernel, kernel modules, rootfs and U-Boot, if these elements were included at the moment when the project was created.*

*The same way, doing a **make install** installs the complete project, including every element of the project.*

4.4. Run the application

At this point the application is built and maybe installed (this step is not required). The normal process in developing stages is to install the application in the NFS exported directory to have it available to use when booting the target from the net (where the rootfs is mounted via NFS).

If working with a target that boots from flash memory (like the factory default image), the application needs to be transferred using FTP and then run from a shell (serial console or telnet session).

4.4.1. Transfer application via FTP

FTP is the simplest way to transfer the application to the target. The target contains an FTP server installed. The FTP server can be connected to as user **anonymous** or **ftp** without password. The file must be transferred to a folder with write permissions like **/tmp**, which is mountpoint of tmpfs.

1. From the folder where the application is, open an FTP connection to the target.
2. Change directory in the target to **/tmp**.
3. Upload the application with the FTP command **put**.
4. Close the FTP connection with **exit**.

```
~/myFullProject/build/apps/hello_world_c$ ls
hello_world  hello_world.c  Makefile
~/myFullProject/build/apps/hello_world_c$ ftp 192.168.42.30
Connected to 192.168.42.30.
220 (vsFTPd 2.0.5)
Name (192.168.42.30:myuser): ftp
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /tmp
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
226 Directory send OK.
ftp> put hello_world
local: hello_world remote: hello_world
200 PORT command successful. Consider using PASV.
150 Ok to send data.
226 File receive OK.
3208 bytes sent in 0.00 secs (59109.7 kB/s)
ftp> exit
221 Goodbye.
~/myFullProject/build/apps/hello_world_c$
```

4.4.2. Run application

Once the application has been uploaded to the target a shell is needed to execute it. Either the serial console (minicom) or a telnet session from the host can be used.

For example, use Telnet:

1. Telnet to the target's IP address.
2. Change directory to where the application was uploaded.
3. Execute the application from there.

```
$   telnet 192.168.42.30
Trying 192.168.42.30...
Connected to 192.168.42.30.
Escape character is '^]'.



BusyBox v1.2.2 (2007.01.16-12:10+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ # cd /tmp
/tmp # ls
hello_world
/tmp # ./hello_world
Hello World!
/tmp # exit
Connection closed by foreign host.
```

5. Configure the Linux kernel

One of the finest features of Linux is the ability to customize the kernel. This means support for unneeded hardware or services can be removed, resulting in a smaller kernel image. Further, it is possible to choose which parts are embedded in the kernel, and which ones are compiled as loadable modules that can be loaded or unloaded at will, as needed.

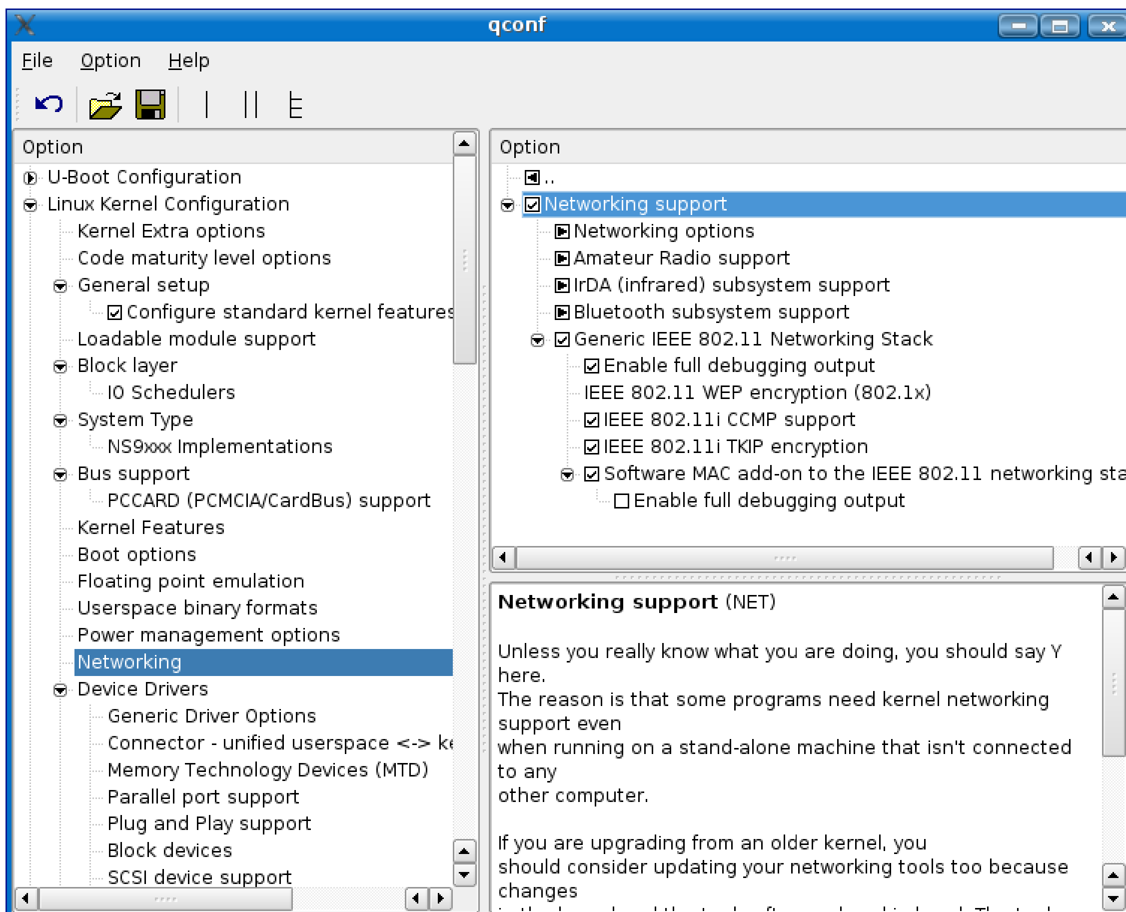
5.1. Kernel configuration options

Customizing the kernel image is done by executing the configuration tool for the project. The project configuration tool is based in the standard kernel configuration tool, that is, a set of Makefile rules used depending on the libraries installed on the development host.

The different options for configuring the kernel are:

```
$   make xconfig (for KDE users - qt libraries required)
$   make gconfig (for GNOME users - gtk libraries required)
$   make menuconfig (graphical configuration tool for console)
$   make config (console configuration tool)
```

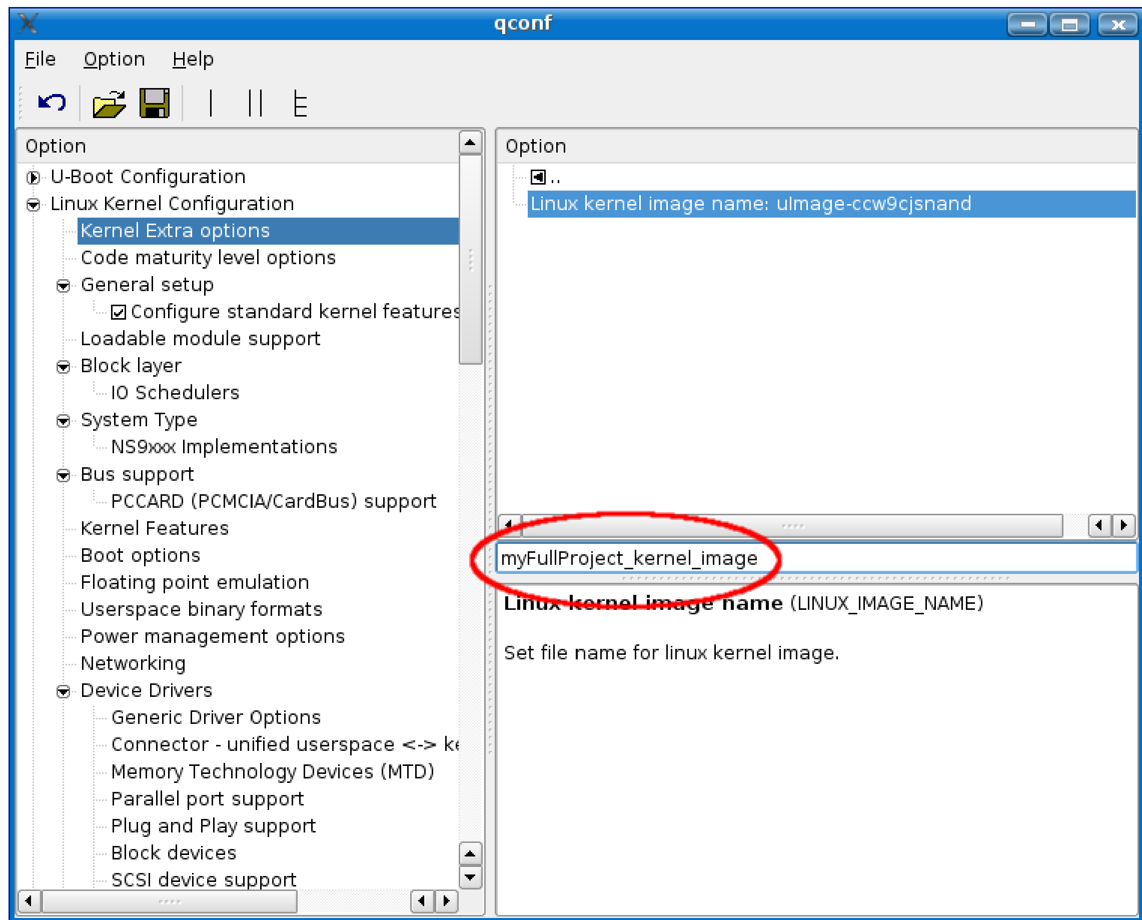
For example, if **make xconfig** is executed, the following window is displayed:



The tool allows customizing the kernel as needed. Most of the options can be built directly into the kernel or as modules (explained in following section).

5.1.1. Kernel image filename

The resulting kernel image filename is, by default **uImage-*target***. This name can be modified at will under the section **Kernel Extra options**.



5.2. Built-in features and kernel modules

Most configuration options are tristate: they can be not built at all (N), built directly into the kernel (Y), or built as a module (M). In the **Linux Kernel Configuration** display, these three states are graphically represented by an unchecked checkbox, a checked checkbox and a checkbox with a black circle in it, respectively.

Built-in features are kernel services or support built directly into the kernel image itself and available as soon as the kernel starts.

On the other hand, *kernel modules* are pieces of code that can be loaded and unloaded into the kernel upon demand. In practice, they are normal files that, when loaded, extend the functionality of the kernel without needing to reboot the system.

While modules are nice for development, or for hardware that might change from time to time, it is recommended that hardware support and kernel features be built directly into the kernel. This way, the kernel can ensure that functionality and hardware support is available whenever it needs it.

For some parts of the configuration, using built-in features is an absolute requirement. For example, if a root file system is a jffs2 file system, the system would not boot if jffs2 support was built as a module. The system would have to look on the root partition to find the jffs2 module, but it cannot look on the root partition unless it already has jffs2 support loaded.

5.3. Platform-specific hardware support

For information about the specific hardware devices, see topic 8, Devices and Interfaces.

5.4. Kernel arguments

The Linux kernel has a monolithic architecture: it is one big program. Arguments are passed on the stack, as in any other C program. The arguments define internal kernel options, such as memory handling, ramdisk handling, root file system handling, and others. Here are some of the most common arguments. For a complete list of kernel arguments, refer to the Linux documentation at **/usr/local/DigiEL-4.0/kernel/linux/Documentation/kernel-parameters.txt**.

Argument	Description	Examples
root=	Tells the kernel which device is used as the root file system while booting.	root=/dev/ram
rootfstype=	Specifies a comma-separated list of file system types that are tried for a match when trying to mount the root file system.	rootfstype=nfs
ro	Instructs the kernel to mount the root file system as **readonly.**	ro
rw	Instructs the kernel to mount the root filesystem as **read/write**.	rw
nfsroot=	Tells the kernel which machine, which directory, and NFS options to use for the root file system. Note that the argument **root=/dev/nfs** is required.	nfsroot=192.168.42.1:/exports
ip=	When using NFS as a root filesystem, there are no programs like **ifconfig** and **route** present until the rootfs is mounted, meaning that the kernel has to configure the network interfaces directly. This argument sets up the various network interface addresses required to communicate over the network. If this argument is not specified, the kernel tries to use RARP and/or BOOTP to determine these parameters.	ip=192.168.42.30
console=	Selects the device to be the console.	console=ttyS1,9600
mtdparts=	Defines the partition table of an MTD device (similar to Flash memory). Configurable options include the partition name, size, offset, and write permissions.	mtdparts=myFlash:0x40000(u-boot)ro, 0x2000000(kernel),-(rootfs)

All the above arguments except console are set automatically by the 'dboot linux ...' command, which overwrites user settings. This command is covered in topic 7.3.

To modify kernel arguments, edit the U-Boot environment variable **bootargs**. Topic 7.4.3 explains some basic U-Boot commands for editing environment variables.

5.5. Kernel modules

Besides selecting which kernel features are compiled as loadable modules, custom module kernel modules can be created, for example, to access special hardware for which a driver does not exist.

5.5.1. Write kernel modules

Writing kernel modules is, from the project perspective, very similar to writing applications. First a project with kernel modules support is needed, (**-m** option in the project wizard script). Since the previously created project **myFullProject** was created with that option, a kernel module can be created in it.

Provide the module's source code and the Makefile in its own folder inside **build/modules/** project subfolder. There are already some example modules inside **build/modules/** folder which can be used as templates. To do this, follow this little instructions:

1. Create a subfolder of **build/modules/** with the name of the module.
2. Create the source files inside the new folder.
3. The Makefile must be provided as well. The Makefile uses the standard rules for Linux Kernel modules. Therefore, if the module source file is **gpio.c** (included as example), the Makefile should contain:

```
obj-m := gpio.o
```

If this interface (the source code + Makefile) is provided in its own directory inside the **build/modules** folder, the topdir project Makefile will be able to build the necessary external kernel modules.

Writing kernel modules is beyond the scope of this manual. A good reference book is "Linux Device Drivers 3rd edition" by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman (ISBN: 0596005903).

5.6. Build the kernel and kernel modules

Once the kernel is configured and any external kernel modules are written, the kernel is ready to be built. To build the kernel and kernel modules execute the following rules in project folder:

```
$   make build_kernel
$   make kernel_modules
```

The first rule builds the kernel, the second builds the kernel modules (any part of the kernel configured to be built as a module and also the custom external kernel modules created in **build/modules/** folder).

After the build, the kernel image is stored in the images subfolder of the project, and the built modules are in their own folders with **.ko** extension (kernel parts inside the **build/kernel/** and the external modules inside the **build/modules/**).

```
$   ls images
uImage-ccw9cjsnand
$   ls build/modules/gpio/
built-in.o  gpio.c  gpio.h  gpio.ko  gpio.mod.c  gpio.mod.o  gpio.o   Makefile
Modules.symvers
```

Now, the kernel image and the modules are ready. The next topic shows how to install them.

*Doing a **make** without any arguments builds the complete project, including applications, kernel, kernel modules, rootfs and U-Boot if these elements were included at the moment when the project was created.*

5.7. Install the kernel

Installing the kernel means copying the resulting kernel image to a place accessible by the target platform.

After the build, the kernel image is stored in the **images/** project subfolder. The target, however, gets the kernel image from TFTP, both to boot or to update the flash memory with the new kernel image. This is why the kernel image must be copied to the exposed folder of the TFTP server. This folder was passed as an option to the project maker wizard at project creation time, so it does not have to be done manually.

To install the kernel image do (as normal user):

```
$   make install_kernel
Installing uImage-ccw9cjsnand
```

This transfers the kernel image to the exposed TFTP directory, making it available to the target via TFTP. Additionally, if the project was created with support for rootfs, the kernel modules will be copied to the proper places in the rootfs, inside **build/rootfs/lib/modules/** folder.

*Doing a **make install** installs the complete project, including every element of the project.*

5.8. Load kernel modules

Kernel modules are files with the extension **ko**. They can be managed on the system with the following commands:

Command	Description
lsmod	Displays the status of the modules in the Linux kernel.
insmod *filename*	Loads a module in the Linux kernel.
rmmod *modulename*	Unloads a module from the Linux kernel.
modprobe *modulename*	Loads a module in the Linux kernel.

For more information on these commands, see their Linux man pages.

Transfer the kernel module (**gpio.ko**, for example) to the target via FTP or, if the target mounts the rootfs via NFS, simply copy the module to the exported directory. Once the kernel module has been transferred to the target, the above commands can be used to load/unload it:

```
#   insmod gpio.ko
gpio: NS9XXX GPIO driver V1.0
#   lsmod
Module              Size  Used by    Not tainted
gpio                5160  0
#   rmmod gpio
#   lsmod
Module                    Size  Used by    Not tainted
#
```

If the project was created with rootfs support, the modules will be installed into the rootfs. They do not need to be copied. Instead, they can simply be loaded, using the **modprobe** *command and the name of the module.*

This GPIO driver provides an interface to configure and manage the GPIO pins of the NS9360 microprocessor. For more information, see topic 8.2.

5.9. Modify kernel sources

Kernel sources are located in a common directory of the Digi Embedded Linux installation folder, **/usr/local/DigiEL-4.0/kernel/linux**.

When a project that includes the kernel component is built, only the generated object files and the final kernel image are stored in the project folder. The kernel sources are not copied to the project folder, thus saving a lot of hard disk space.

Suppose the kernel sources will be modified to add certain functionality or customizations to a driver. If the kernel sources were directly edited, those changes would apply to all projects including the kernel component. However, unless the changes are well-known kernel patches, that is not normally desired.

To edit the Linux kernel locally to a project copy the files to be modified to the local project's Linux kernel folder, located at ***your_project_path*/build/kernel**, with exactly the same directory structure that the copied files have in the original kernel directory tree.

For example, to modify the kernel init process, and specifically the file **/usr/local/DigiEL-4.0/kernel/linux/init/main.c**:

1. Create the path in the **build/kernel/** directory of the project
2. Copy the original kernel file to the project kernel folder.

```
$   mkdir -p build/kernel/init
$   cp /usr/local/DigiEL-4.0/kernel/linux/init/main.c build/kernel/init/
```

3. Edit and modify the local kernel file. During compilation, the environment will check for local files first and, if they exist, it will compile these sources instead of the global kernel sources.

5.9.1. Import all kernel sources

Obviously, all kernel sources can be imported to a local project, providing a full local copy of the kernel. This requires much more hard disk space, but allows working on the entire kernel source code safely.

To do this, copy all kernel sources to the project's **build/kernel/** folder.

```
$   cp -r /usr/local/DigiEL-4.0/kernel/linux/. build/kernel/
```

The copy process can take several minutes because the kernel sources occupy around 285 Mb.

Changes done in the project kernel folder are compiled in the build process.

6. Customize the root file system

This topic shows how to customize the rootfs of the target. It uses the full project, **myFullProject**, created in previous topics.

6.1. Configure the rootfs

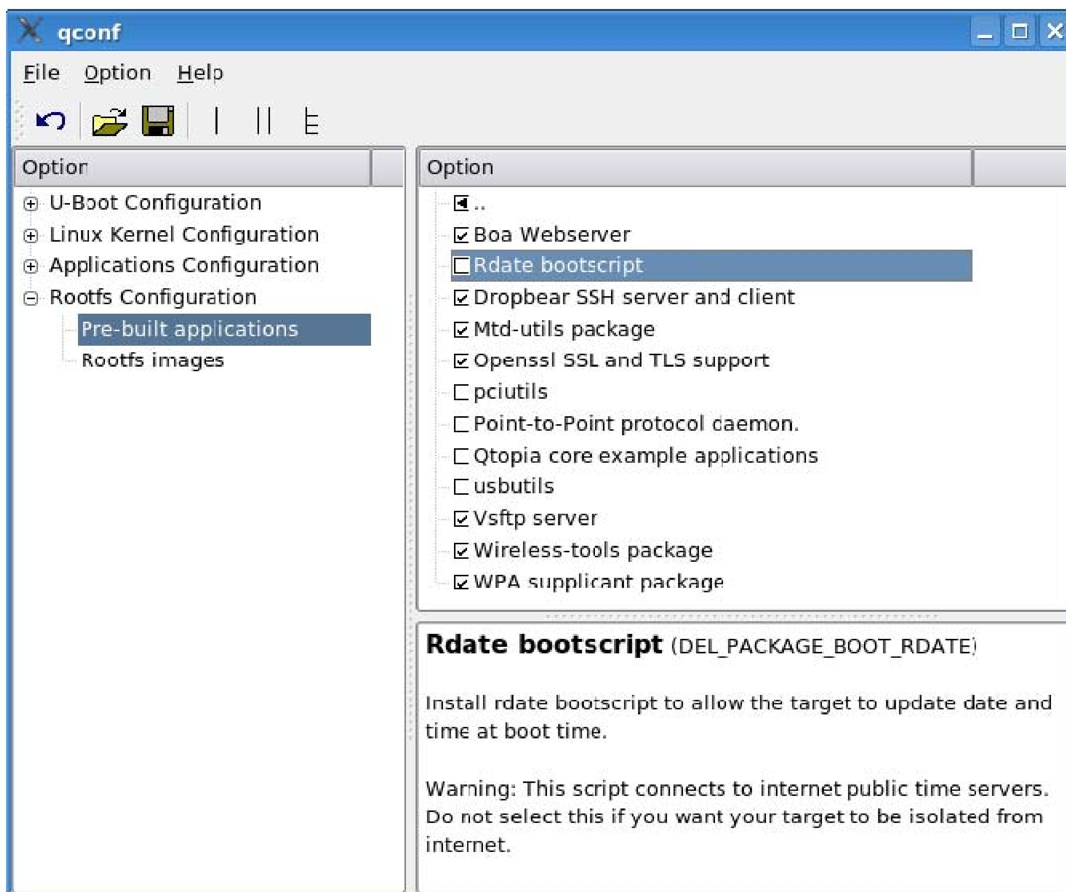
Once the project is created, a basic rootfs is in folder **build/rootfs/**. To configure the rootfs, use the **Configure Project** tool seen in previous topics. Just use one of the following:

```
$   make xconfig (for KDE users - qt libraries required)
$   make gconfig (for GNOME users - gtk libraries required)
$   make menuconfig (graphical configuration tool for console)
$   make config (console configuration tool)
```

The Configuration tool is opened. Expand the **Rootfs Configuration** for configuring the rootfs. For the Root File System, there are two configuration menus: **Pre-built applications** and **Rootfs images**.

6.1.1. Including pre-built applications

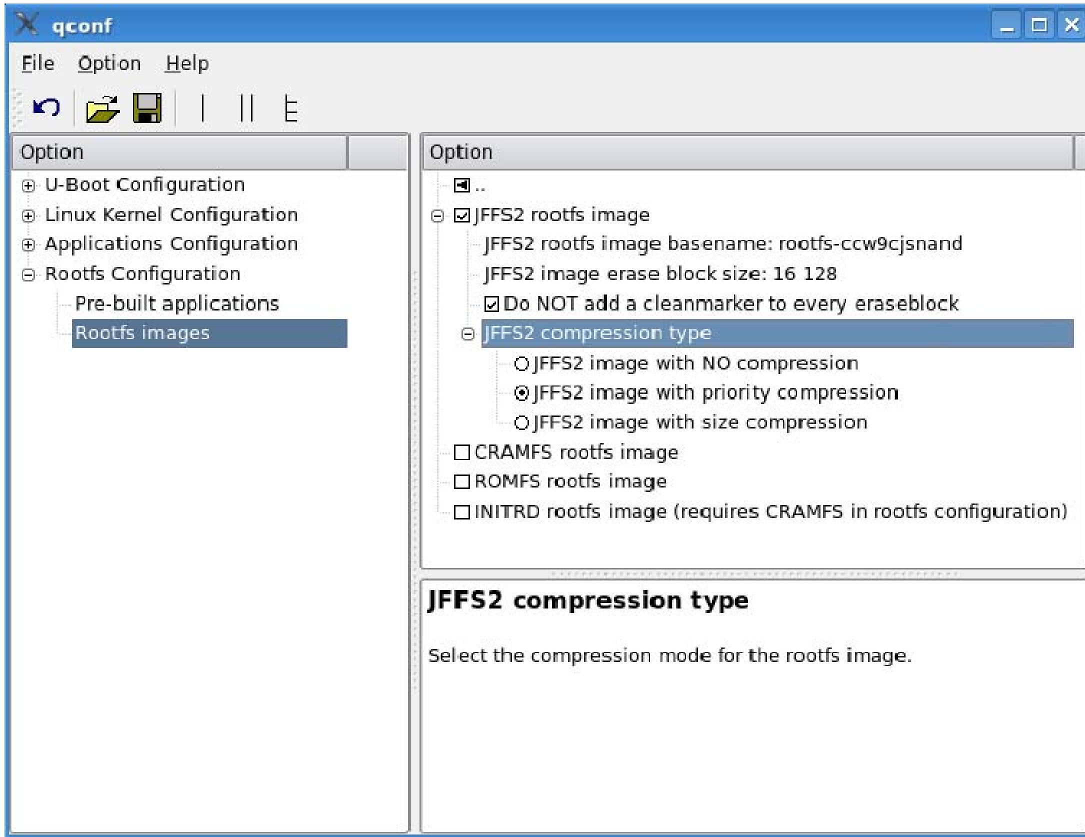
The pre-built applications menu selects different applications to include in the rootfs.



All the source code for these applications is included in the liveDVD, in folder **/toolchain**. Pre-built applications and services are briefly explained in topic 6.6.

6.1.2. Rootfs images

The Rootfs images are different file systems that can be used as the root file system:



6.1.2.1. Network File System (NFS)

A Network File System (NFS) acts like a server/client application that lets a system use files stored on a remote computer as if they were in the local one. The system using the files requires an NFS client, while the system serving the files requires an NFS server. Both of them require the TCP/IP stack for communication. For information about how to configure the NFS server, see topic 2.3.2.

6.1.2.2. Journaling Flash File System 2 (JFFS2)

A Journaling Flash File System 2 (JFFS2) is a log-structured file system specially designed with flash devices in mind and also the ability to be consistent after a sudden power outage. For more about JFFS2, see **http://en.wikipedia.org/wiki/JFFS2**

6.1.2.3. Compressed ROM filesystem (CRAMFS)

A Compressed ROM filesystem (CRAMFS) is a read-only file system designed for simplicity and space-efficiency. It is mainly used in embedded systems and small-footprint systems. For more about CRAMFS, see **http://en.wikipedia.org/wiki/Cramfs**

6.1.2.4. ROM filesystem (ROMFS)

A ROM filesystem (ROMFS) is a space-efficient, small, read-only filesystem for Linux. For more about ROMFS, see **http://romfs.sourceforge.net/**

6.1.2.5. Initial ramdisk filesystem (INITRD)

The initial ramdisk (or INITRD) is a temporary filesystem used by a Linux kernel during boot. This configuration option provides an initrd image of the rootfs. For more about INITRD, see **http://en.wikipedia.org/wiki/Initrd**

6.1.3. Report the rootfs type to the kernel

The type of rootfs must be passed as an argument to the kernel. This is done by editing the kernel argument **rootfstype** in the U-Boot variable **bootargs** as seen in topic 5.4.

6.2. Put files and folders in the rootfs

Once the rootfs is configured with the pre-built applications and the image types, add folders and files to the rootfs structure.

There are two ways to add files and folders:

- Modifying the **rootfs** folder
- Using the **add_files.sh** script

6.2.1. Modify the rootfs folder

The target's root file system is populated in **build/rootfs/** folder. Custom files and folders can be copied and created in that location and will be available after the build and installation process.

*Changes in the **build/rootfs/** folder require a build and install to make them available to the target.*

This method has a disadvantage: executing a **make clean** or a **make rebuild** completely erases and regenerates the **build/rootfs/** folder, and any files or folders that were created are lost.

6.2.2. The add_files.sh script

The **add_files.sh** script is in the folder **configs/** of the project directory. This script is initially empty, but can be populated with commands to create folders and copy files to the **build/rootfs/** folder. The **add_files.sh** script is called during the build process, thus constructing the desired rootfs at compilation time. This script can use all the power of the Linux host shell, such as create directories, change permissions, copy files, use conditions and loops, etc. It also inherits several environment variables and has defined one, **ROOTFS_DIR**, that points to the **rootfs** folder, for ease of use. Here is an example of a basic **add_files.sh** script:

```
ROOTFS_DIR="${DEL_PROJ_DIR}/build/rootfs"

## Example: create a custom directory in rootfs etc dir.
mkdir -p "${ROOTFS_DIR}/etc/myfolder"
mkdir -p "${ROOTFS_DIR}/etc/myimgs"

## Example: copy files to a directory
cp ~/*.jpg "${ROOTFS_DIR}/etc/myimgs"
```

6.3. Build the rootfs

The rootfs is ready to be built. Building the rootfs means creating the final functional rootfs, depending on the options selected at the configuration stage. It also creates the images for the selected rootfs types.

To build the rootfs, execute:

```
$   make build-rootfs
```

This action creates the rootfs, checks library dependencies, and creates the images. These images are stored in the **images/** subfolder.

Doing a ***make*** *without any arguments builds the complete project, including applications, kernel, kernel modules, rootfs and U-Boot if these elements were included at the moment when the project was created.*

6.4. Install the rootfs

At this point the rootfs has been created, configured, and built. The final stage is installing the rootfs, which makes the rootfs available for the target to find it. This is done by copying the rootfs images to the TFTP-exposed directory, and copying a rootfs tree to the NFS exported directory. Both parameters were passed as an option to the wizard script at project creation time (see topic 3.2.6).

To install the rootfs, execute:

```
$   make install_rootfs
```

Now the target has a rootfs to boot via NFS, or rootfs images to update the flash memory.

Doing a ***make install*** *installs the complete project, including every element of the project.*

6.5. Special files

The **/etc** folder contains configuration files for applications, services and the system itself. Some of the files in the **/etc** folder include:

File	Description
/etc/inittab	Parameters for the init process, the first process started at boot time.
/etc/fstab	A list of file systems to be mounted.
/etc/init.d/*	System startup and run level change scripts.
/etc/passwd	Critical list of users, home directories, etc.
/etc/shadow	User passwords.
/etc/group	A list of groups of the system.
/etc/vsftpd	FTP daemon configuration file.
/etc/boa/boa.conf	BOA web server configuration file.
/etc/dropbear/*	SSH server/client configuration files.
/etc/makedevs.conf	Configuration table to create device nodes automatically at startup.

6.6. Built-in applications and services

By default, the rootfs contains many folders and files. Some of these files are useful applications and services. They can be included as needed in the final rootfs. All the source code for these applications is included in the liveDVD, in folder **/toolchain**.

6.6.1. busybox

The busybox application provides many Linux utilities, such as **cat**, **chmod**, **echo**, **mount**, **sh** and others. All these tools have been linked into the static application **busybox** to save Flash memory at the cost of a larger RAM footprint for each application.

6.6.1.1. Useful network tools provided in busybox

busybox provides some useful network tools, such as:

- **wget**: Retrieves files via HTTP or FTP
- **ftpget**: Retrieves a remote file via FTP.
- **ftpput**: Stores a local file on a remote machine via FTP.

To display brief help text on the commands in Busybox, enter ***command*** **--help**, for example:

wget --help
ftpget --help
ftpput --help

6.6.2. FTP server

By default, an FTP server is included in the rootfs. The server allows other machines to establish an FTP session to the target device. For configuration options and further details, see the vsftpd documentation at **/usr/local/DigiEL-4.0/docs/Software/Packages/vsftpd-2.0.5**. The FTP server can be connected to as user **anonymous** or **ftp** without password. Or log in as user **root** and password **root**.

6.6.3. Boa webserver

A small Boa webserver is included and running in the default rootfs. By default, it serves any web page located at folder **/var/www/** of the rootfs. For configuration options and further details, see the Boa documentation at **/usr/local/DigiEL-4.0/docs/Software/Packages/boa-0.94.14rc21** or the official Boa website at **http://www.boa.org**

6.6.4. gdbserver

gdbserver is a server that allows GDB debugging sessions to be established between a development machine and the device target. For more information about GDB, see **http://sourceware.org/gdb/documentation/**

6.6.5. update_flash

update_flash is a custom application that allows writing to the Flash partitions from Linux. The graphic wrapper of this tool is explained in topic 7.4.

6.6.6. nvram

The **nvram** tool is a custom application for managing settings in NVRAM, such as MAC addresses, IP configuration, the Flash partition table, and OS-specific configuration. It is the same application as the **intnvram** U-Boot command. For more information about NVRAM settings and the **nvram**

application, see the topic *Using NVRAM* in the *U-Boot Reference Manual* or type "nvram --help" in the target Linux system.

6.6.7. boot_rdate

boot_rdate is a bootscript that sets the system's date and time from a remote host at boot time. It uses the **rdate** utility (part of **busybox**) and a list of hosts configured in **/etc/rdate.conf** file.

6.6.8. pppd

The Point-to-Point Protocol daemon (**pppd**) is used to establish network connections between two nodes. For more information on this daemon, see the doc at **/usr/local/DigiEL-4.0/docs/Software/Packages/ppp-2.4.4** or visit **http://ppp.samba.org/ppp/index.html**

6.6.9. mtdutils

mtdutils contains utilities to manipulate memory technology devices, such as flash memory. For more information about **mtdutils**, see each tool's man page at **/usr/local/DigiEL-4.0/docs/Software/Packages/mtd-utils-1.0.0** or visit the mtdutils homepage at **http://www.linux-mtd.infradead.org/index.html**

6.6.10.wireless-tools

wireless-tools contains a set of tools for configuring wireless interfaces, access points, encryption, etc. on wireless LANs (WLANs). For more information on the wireless-tools, see their man-pages located at **/usr/local/DigiEL/docs/Software/Packages/wireless_tools-28** or the wireless-tools homepage at **http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html**

6.6.11.dropbear SSH

dropbear is a small SSH server and client. Because of its small memory footprint, **dropbear** is particularly useful for embedded Linux systems. To connect to the target's IP use username **root** and password **root**:

```
$   ssh root@192.168.42.30
```

For more information about **dropbear**, see the doc at **/usr/local/DigiEL-4.0/docs/Software/Packages/dropbear-0.48.1** or visit its homepage at **http://matt.ucc.asn.au/dropbear/dropbear.html**

6.6.12.udhcp server/client

udhcp server/client is an embedded DHCP server/client package that strives to be fully functional, RFC compliant, and very tiny. This makes **udhcp** useful for embedded systems. For more information about udhcp, see its homepage at **http://udhcp.busybox.net**

6.7. Launch an application after start-up

When the kernel finishes loading, it tries to mount the rootfs. When the rootfs is mounted, the **init** process runs and forks additional processes as specified in **/etc/inittab** file. This file contains a line to launch the script **/etc/init.d/rcS** which eventually launches any other script in this folder.

To launch an application automatically after start-up, a launch script must be provided for the application inside the **/etc/init.d/** directory of the rootfs.

The script's name must follow this pattern: **S??*name*.sh**. This means begin the name with a capital S, followed by two numbers that define the order of execution at boot time, followed by a descriptive name, and the extension **sh**. For example: **S11myscript.sh**, **S34myOtherScript.sh**, **S99final.sh**.

The lower the number in the script name, the sooner the script executes.

All the scripts stored in folder **/etc/init.d/** are executed at start-up with **start** parameter. Usually, start-up scripts define different actions to perform if the parameter is **start** or **stop**, but it is not mandatory to provide such interface. Just create a script that follows the naming pattern requirement, and it will be executed at startup.

There are already some start-up scripts in that folder, such as **S04makedevs.sh**, **S10rdate.sh**, etc, than can be used as examples for creating a start-up script for the application.

Remember that any changes to a project's **build/rootfs/** *folder do not apply until the rootfs is built and installed again.*

7. Transfer the system to the target

Up to this point in the example shown in this document, these elements have been generated::

- kernel image
- root file system image
- applications

As shown in topic 4.4.1, applications are easily transferred to a running target, by means of network services like FTP.

This topic shows how to transfer the kernel and root file system images to the target for testing prior to updating the firmware in the Flash memory.

7.1. Basic boot loader commands

Testing the recently created kernel and root file system requires working with several boot loader commands. This topic explains the basic commands to download and boot a new system. For more information on U-Boot commands, see the *U-Boot Reference Manual*.

Power up the target board. When the boot loader messages are displayed in the Serial Console, press a key to stop the auto boot process.

```
U-Boot 1.1.4 (Feb 20 2007 - 14:23:03) DEL_4_0_RC3
for Digi ConnectCore Wi-9C on Development Board

DRAM:   64 MB
NAND:   128 MiB
In:     serial
Out:    serial
Err:    serial
CPU:    NS9360 @ 154.828800MHz
Strap: 0x03
SPI ID:2007/01/25, V1_4rc2, CC9C/CCW9C, SDRAM 64MByte, CL2, 7.8us, LE
FPGA:   wifi.ncd, 2007/01/25, 17:49:41, V2.01
Hit any key to stop autoboot:  0
#
```

These U-Boot commands will be used:

Command	Description
printenv [variable]	Displays all (or one) current variables values.
setenv <variable> <value>	Sets a U-Boot variable with a given value.
saveenv	Saves all U-Boot variables into the NVRAM memory for permanent storage.
dboot <os> <type>	Boots an image.

7.2. U-Boot variables

The U-Boot boot loader contains several variables that configure its behavior. Some of the variable values need to be modified.

Before modifying U-Boot variables, check their values with the 'printenv' command, because they might already be properly configured.

Since the transfer of the images takes place over Ethernet, first set proper values for the Ethernet parameters, such as the IP address of the target and the host. This example uses an IP address of **192.168.42.30** for the target board and **192.168.42.1** for the development computer (host).

```
#   setenv ipaddr 192.168.42.30
#   setenv serverip 192.168.42.1
```

Another variable that needs to be set up is the path to the NFS exported folder where the root file system resides. In this example, the path is the default, **/exports/nfsroot-ccw9cjsnand**:

```
#   setenv npath /exports/nfsroot-ccw9cjsnand
```

These variables are modified in RAM. Optionally, to save them permanently in the NVRAM memory, use the **saveenv** command:

```
#   saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
```

7.2.1. Changing U-Boot variables

U-Boot variables in the target can also be changed from Linux by means of the **ubootenv** application. Here is the **ubootenv** syntax:

```
#   ubootenv -h
Usage: ubootenv [options]
ubootenv $Revision: 1.4 $ Copyright Digi International Inc.

Prints or updates the U-Boot environment

   -d, --dump                     Prints the values of all the
                                  environment
   -p, --print 'var_name_list'    Prints the value of the list of
                                  variables
                                  The list has to be simple quoted ('')
   -s, --set  'var_name=var_value'  Sets var_value in the variable var_name
                                  The string has to be simple quoted ('')
                                  to allow spaces
   -e   --erase 'var_name_list'   Removes the list of variables (simple
                                  quoted)
   -c   --clean                   Removes all variables
   -a   --fileadd file_name       Adds variables from file_name. To init
                                  the full environment from file use -c
                                  -a simultaneously
   -h   --help                    Displays usage information
```

7.3. Test the system

Next, the system is transferred to the target and booted it from RAM memory. This allows testing the system without needing to update the Flash memory.

7.3.1. Transfer the system by Ethernet

In previous topics, the kernel image was installed in the exposed TFTP folder and the root file system in the exported NFS folder. Now, the boot loader will be instructed to download the kernel via TFTP and mount the rootfs via the NFS server.

Now that there is a proper Ethernet configuration in U-Boot, it can be instructed to download the kernel, boot it, and mount the rootfs as NFS. These three steps are done with a single command:

```
#   dboot linux tftp
```

7.3.2. Transfer the system by USB

U-Boot can also read the kernel from a USB flash disk. In this case, the kernel image must be copied to a USB flash disk formatted with FAT file system. Then, plug the USB to the target, power up the board and stop the U-Boot auto boot.

A valid JFFS2 rootfs image must reside in the RootFS partition in flash. Only the kernel will be taken from the USB disk.

Now U-Boot is instructed to read the kernel from the USB disk, boot it, and mount the rootfs from flash memory RootFS JFFS2 partition. These three steps are done with a single command:

```
#   dboot linux usb
```

7.4. Update the Flash memory

The kernel and rootfs built in previous topics has been tested. This system was dependent on Ethernet/USB to download the kernel and mount the rootfs.

If the tested system behaves correctly and fulfills our needs, it can now be written to the Flash memory of the target, and permanently saved.

There are several ways to update the flash memory depending on what needs to be updated. Applications or simple files, for example, can be simply copied to a file system that resides in a Flash partition. The kernel and the rootfs image, on the other hand, need special handling.

Learning how to update Flash memory requires an overview of its structure.

7.4.1. Structure of Flash

Flash memory is a programmable non-volatile memory thought to contain the whole operating system, thus making the target device a standalone solution.

Flash memory is partitioned, or logically divided, to contain the boot loader, the Linux kernel, the root file system, some system configuration parameters, and some free space to be managed as needed.

The number, size and position of these partitions can be modified as needed. The factory default partitioning structure is as follows:

Partition number	Name	Flash start address	Flash end address	Length	Description
0	U-Boot	0x00000000	0x000c0000	768 KB	Stores the U-Boot boot loader image.
1	NVRAM	0x000c0000	0x00100000	256 KB	Stores permanent configuration parameters such as the MAC address of the network interfaces, the serial number of the module, environment variables of U-Boot, etc.
2	FPGA	0x00100000	0x00200000	1 MB	Stores the FPGA firmware.
3	Kernel	0x00200000	0x00500000	3 MB	Stores the Linux kernel image.
4	RootFS-JFFS2	0x00500000	0x01500000	16 MB	Stores the Linux root file system image.
5	User-JFFS2	0x01500000	end of flash	rest	Free space.

For information about modifying the default Flash partition table, see the topic *"Using NVRAM"* in the *U-Boot Reference Manual*.

7.4.2. Update from a running Linux system

Digi Embedded Linux provides a tool called **update_flash**, which can directly write image files to the Flash memory if the target is already running a Linux system. It is more powerful than the standard Linux **flash_write** or **flash_erase** as it checks whether the images are correct and suitable for a certain partition. On NAND memories, bad sectors are also handled.

Here is the **update_flash** command syntax:

```
#   update_flash --help
Usage: update_flash [--version] [--help] [--log-level] [--checksum-only] [--
reboot] [--verify-only] [--verify] [--checksum] [--dry-run] [--progress-in-
new-line] [--no-image-type-check] [--erase-all] [--clean-marker] [--silent]
[--tmpdir] <file [part] [checksum]>
    version                          : print version and exit
    help                        [-h] : print help
    log-level                   [-l] : log level for messages
    checksum-only               [-C] : calculates only CRC32 checksum of image
    reboot                      [-R] : reboots the system
    verify-only                 [-V] : verifies current contents, no updates
                                       are done
    verify                      [-v] : After flashing, compare flash contents
                                       with image on byte-to-byte
    checksum                    [-c] : flashes only when checksum matches
    dry-run                          : don't erase or write to the flash
    progress-in-new-line             : each percentage is printed in an own
                                       line
    no-image-type-check         [-i] : doesn't checks image type for part. 0-2
    erase-all                   [-f] : erases the partition, not only the parts
                                       being written. Ignored for rootfs
    clean-marker                     : writes clean markers to every partition
                                       (implies -f)
    silent                      [-s] : Silent Mode
    tmpdir                      [-t] : copy files to temporary directory before
                                       flashing
    <file [part] [checksum]>         : file to flash to partition and check for
                                       checksum

Flash Update Tool

Examples of use cases:
  update_flash rootfs-ccw9cjsnand-128.jffs2 4 : updates rootfs
  update_flash -C uImage-ccw9cjsnand : calculate CRC32 only
  update_flash -c uImage-ccw9cjsnand 3 0x1051e3c9 :
               update only if CRC32 is 0x1051e3c9
  update_flash uImage-ccw9cjsnand 3 rootfs-ccw9cjsnand-128.jffs2 4:
               update both kernel and rootfs
```

7.4.2.1. Update the kernel

First, the new kernel image must be accessible from the target side. This can be done by having it stored in a USB Flash disk or in a Compact Flash card, by transferring it via FTP or by mounting an NFS folder.

Then, run the **update_flash** command with the correct parameters. For example, if the new kernel image is the file **/nfs/uImage-ccw9cjsnand** and the kernel partition is number **3**, execute:

```
#   update_flash uImage-ccw9cjsnand 3
Partition 3 is NAND (Kernel)
 Full Size: 3072 KiB
 Good Size: 3072 KiB
Verifying File(s):   uImage-ccw9cjsnand
Updating:
 uImage-ccw9cjsnand (1252 KiB)
 Erasing:   complete
 Flashing:  complete
 CRC32:     0xa2a1bf4e
Done
```

The tool checks and reports the type of partition. Then the partition's complete size and the size regarding only good sectors is reported. The file to update is checked whether it fits into the good size. It is also checked whether it can be read completely or whether there are any I/O errors. Also a CRC32 sum is calculated on the file.

When all checks are performed, only the necessary Flash blocks are erased and overwritten to reduce update time. At the end, the image for the Memory Technology Device (MTD), a Linux subsystem for memory devices, is checked against the CRC32 sum for a quick verify.

7.4.2.2. Update the rootfs

Updating the rootfs is similar to updating the kernel. For example, if the new rootfs image is the file **/nfs/rootfs-ccw9cjsnand-128.jffs2** and the rootfs partition is number **4**, execute:

```
#   update_flash rootfs-ccw9cjsnand-128.jffs2 4
```

JFFS2 partitions are always completely erased and marked with clean markers. If the file system is currently mounted, it is unmounted or remounted read-only. When updating the rootfs, it is recommended to do a reboot, as the file system layer of Linux isn't informed about underlying changes.

7.4.2.3. Update the kernel and rootfs safely

The update of the Flash is a critical operation as it erases the Flash where the system resides. If there are errors during the write process, the system cannot reboot Linux. The **update_flash** tool has some options to make the process safer.

The file being written, for example, might reside in an NFS folder and there could be network problems during transmission. To avoid that, copy all files to a temporary directory on the module.

There is an option to do a byte-to-byte verification of the original file and the contents in the Flash partition. To make sure that the files are copied correctly, but also are really the original files, a checksum of the files can be provided.

First calculate the checksums of the files (provided the images are in **/nfs** folder):

```
#   update_flash -C /nfs/uImage-ccw9cjsnand /nfs/rootfs-ccw9cjsnand-128.jffs2
CRC32 Results:
  /nfs/uImage-ccw9cjsnand              : 0xa2a1bf4e
  /nfs/rootfs-ccw9cjsnand-128.jffs2    : 0x2358d051
Done
```

Then execute the **update_flash** command with the parameter **–c** for checking the checksums.

```
#   update_flash -R --tmpdir=/tmp -v -c /nfs/uImage-ccw9cjsnand 3 0xa2a1bf4e
    /nfs/rootfs-ccw9cjsnand-128.jffs2 4 0x2358d051
```

7.4.3. Update from U-Boot

U-Boot is also able to write to the Flash memory. This way, even if the target is not running Linux, the Flash memory can be reprogrammed.

The **update** command updates Flash memory in U-Boot. This is the **update** command syntax:

```
#   help update
update partition source [file]
  - updates 'partition' via 'source'
    values for 'partition': uboot, linux, rootfs, userfs, eboot, wce
                            or any partition name
    values for 'source': tftp, usb
    values for 'file': the file to be used for updating
```

The update command gets the file either from a USB Flash disk or from a TFTP exposed folder in the host, depending on the **source** parameter. For that, it uses the **file** given as parameter or, if no filename is provided, it uses the names stored in the following U-Boot environment variables:

- Kernel image filename: **kimg**
- Rootfs image filename: **rimg**
- User image filename: **usrimg**
- U-Boot image filename: **uimg**

The default values for these variables correspond to the default image filenames generated during compilation of the system. If the image filenames were changed, provide the parameter **file** with the new name to the **update** command.

The U-Boot **update** command takes care of transferring the image file to RAM, erasing the Flash sectors, and writing the new image.

There are some restrictions when updating large image files. See topic 12.3 for more information.

7.4.3.1. Update the kernel

For example, if the kernel is in a TFTP exposed folder on the development computer, the **update** command is:

```
#   update linux tftp
```

7.4.3.2. Update the rootfs

Updating the rootfs is similar to updating the kernel. For example, if the new rootfs image is stored in a USB flash disk, the **update** command is:

```
#   update rootfs usb
```

7.5. Boot from Flash memory

Now that the system has been transferred to the Flash memory, the boot loader can be instructed to boot directly from Flash. This is done with the following U-Boot command:

```
#   dboot linux flash
```

To boot from Flash automatically, the **bootcmd** U-Boot environment variable must be modified.

```
#   setenv bootcmd dboot linux flash
#   saveenv
```

Now, the target will automatically boot directly from the Flash memory.

8. Devices and Interfaces

This topic explains the different devices and interfaces available in the hardware platform, the hardware resources they use, how to configure them, how to enable or disable them, how are they seen by the system, and how to manage them from user application space.

8.1. Table of devices and their hardware resources

This table shows each device/interface with its driver name and the hardware resources it uses.

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
System		16		0-0x20000000	16	dynamic 0
GPIO	gpio		0-72 (muxed)			
Ethernet	ns9xxx_eth	4,5	50-64			
NAND	ccx9x_nand			0x50xxxxxx		static 1
Serial A (UART) /dev/ttyS1	ns9xxx_serial	36,37	8,9 (10-15)[1]			
Serial B (UART) /dev/ttyS0	ns9xxx_serial	34,35	0,1 (2-7)[1]			
Serial C (UART) /dev/ttyS2	ns9xxx_serial	38,39	40,41 (42,43, 20-23)[1)]			
Serial D (UART) /dev/ttyS3	ns9xxx_serial	40,41	44,45 (46,47, 24-27)[1]			
Serial A (SPI mode)	ns9xxx_spi	60,61	8,9,14,15			
Serial B (SPI mode)	ns9xxx_spi	58,59	0,1,6,7			
Serial C (SPI mode)	ns9xxx_spi	62,63	40,41,22, 23			
Serial D (SPI mode)	ns9xxx_spi	64,65	44,45,26, 27			
Touch screen (SPI Port B)	ads7846	EXT 3	0,1,6,7			
I²C	i2c-ns9xxx		46,47			
I²C I/O port PCA9554	pca9554					
RTC	ns9360_rtc					
USB		25				
Wireless	digi_wi_g	65	58,65-67	0x6xxxxxxx		static 2
Display	ns9xxxfb		15,18-41			
high performance counter[2]					1	

[1] Only when HW Handshaking is enabled

[2] Only available as an include file for drivers (**ns9xxx_hperf.h**).

To determine the live information of the running system, use the following Linux commands:

Command	Information
cat /proc/interrupts	Interrupts used by drivers.
cat /proc/ioports	GPIOs used by drivers.
cat /proc/iomem	Physical Memory used by drivers. Range 0x90000000 to 0xafffffff are chip's registers.

Most drivers reserve GPIOs, interrupts, and memory only when being used. This means Serial Port C and LCD cannot be used at the same time because they share some multiplexed GPIOs.

8.2. GPIO pins and custom driver

The NS9360 processor has a total of 73 programmable GPIO pins, multiplexed with other functions. A custom driver for configuring and managing the GPIO pins is provided in the Digi Embedded Linux software. The source files for this custom driver are included into the kernel modules folder, in **/usr/local/DigiEL-4.0/modules**.

8.2.1. Hardware resources used by the driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select	DMA channel
GPIO	gpio		0-72 (muxed)				

8.2.2. Enable the interface in the kernel

The GPIO driver is provided as an external kernel module. A project with support for kernel modules (see topic 3.2.2). is required to include this module into the filesystem. These projects have the GPIO driver sources automatically included into **build/modules/gpio/** folder.

After compiling the kernel and modules, a binary file **build/modules/gpio/gpio.ko** is created.

For projects created with support for rootfs, the **gpio** module is placed in the rootfs **lib** directory, where appropriate. For projects that do not have rootfs support, this file must be placed somewhere in the target's rootfs.

8.2.3. Load and create the device in the system

8.2.3.1. Load the module

To load the GPIO module use the **insmod** or **modprobe** commands without any arguments, as seen in topic 5.8.

The **modprobe** command can be used when the project was created with rootfs support, because it creates the module's dependencies file within the rootfs. In the **modprobe** command, only the driver name must be specified:

```
#   modprobe gpio
NS9XXX GPIO driver V1.0
```

If a project was created without rootfs support and the module was copied manually, the module must be loaded using the **insmod** command, specifying the complete path and filename of the module. For example, if the module has been copied to the **/tmp** folder, the **insmod** command to load the module is:

```
#   insmod /tmp/gpio.ko
NS9XXX GPIO driver V1.0
```

8.2.3.2. Create device nodes

Device nodes must be manually created, one per GPIO pin. First, create a container folder named **/dev/gpio**. Then add to that location a character node with major number **250** and a minor number that matches the GPIO number. For simplicity, this example names the GPIO pins by their numbers, as shown below:

```
#   mkdir /dev/gpio
#   cd /dev/gpio
#   mknod 48 c 250 48
#   mknod 53 c 250 53
#   mknod 60 c 250 60
#   ls -l
crw-rw----    1 root     root      250,  48 Jan  1  1970 48
crw-rw----    1 root     root      250,  53 Jan  1  1970 53
crw-rw----    1 root     root      250,  60 Jan  1  1970 60
```

Alternatively, to have the system create the device nodes after startup, edit the special system file **/etc/makedevs.conf** with entries for creating the **gpio** folder and the required device nodes. For example:

```
# Example:
#<name>    <type><mode><uid><gid><major><minor><start><inc><count>
/dev/gpio       d    755
/dev/gpio/48    c    660 0    0    250      48
/dev/gpio/53    c    660 0    0    250      53
/dev/gpio/60    c    660 0    0    250      60
```

8.2.4. Manage GPIO pins from the user space

The GPIO driver exports several functions to manage the GPIO pins from the applications. To use these functions, applications must include the driver's header file, **gpio.h**, which contains some GPIO driver definitions.

8.2.4.1. Device IO Controls

The driver supports the following I/O controls (IOTCLs). Parameters are passed as pointers to integers.

IOCTL	Description	Parameter
GPIO_CONFIG_AS_INP	Configures the GPIO as input.	N/A
GPIO_CONFIG_AS_OUT	Configures the GPIO as output.	N/A
GPIO_CONFIG_AS_IRQ	Configures the GPIO as IRQ (falling/rising edge or low/high level sensitive).	IRQ_FALLING IRQ_RISING IRQ_LOW IRQ_HIGH
GPIO_CONFIG_INV_PIN	Configures whether the GPIO is internally inverted or not.	0 | 1
GPIO_READ_PIN_VAL	Reads a GPIO configured as input.	variable where to store the value
GPIO_WRITE_PIN_VAL	Writes a GPIO configured as output.	0 | 1

8.2.4.2. Open the device nodes

Since each GPIO pin has its own device node, each one has to be opened separately by calling the standard **open** system call on each device node, as in this example code:

```
int fd48;

if( fd48 = open( "/dev/gpio/48", O_RDWR ) < 0 )
{
    /* ERROR */
}
```

8.2.4.3. Configure GPIO behavior

GPIO pins can be configured to have the following behaviors:

- Input
- Output
- IRQ, which can be:
  - **IRQ_FALLING**: falling edge sensitive
  - **IRQ_RISING**: rising edge sensitive
  - **IRQ_LOW**: low level sensitive
  - **IRQ_HIGH**: high level sensitive

This IOCTL configures the GPIO behavior:

```
/* Configure GPIO as INPUT */
if( ioctl( fd48, GPIO_CONFIG_AS_INP, 0 ) < 0 )
{
    /* ERROR */
}

/* Configure GPIO as OUTPUT */
if( ioctl( fd53, GPIO_CONFIG_AS_OUT, 0 ) < 0 )
{
    /* ERROR */
}

/* Configure GPIO as IRQ, sensitive on falling edge */
ext_irq_type_t irqtype = IRQ_FALLING;
if( ioctl( fd60, GPIO_CONFIG_AS_IRQ, &irqtype ) < 0 )
{
    /* ERROR */
}
```

To configure inversion of the GPIO, use **IOCTL GPIO_CONFIG_INV_PIN**. The address of the variable that contains the inversion value **0|1** is passed as an argument.

```
/* Enable inversion of GPIO */
int inverted = 1;
if( ioctl( fd53, GPIO_CONFIG_INV_PIN, &inverted ) < 0 )
{
    /* ERROR */
}
```

8.2.4.4. Read GPIO inputs

There are two ways to read GPIOs working as inputs.

The first uses the **IOCTL GPIO_READ_PIN_VAL**. The address of the integer variable where the input value is to be stored is passed as an argument.

```
int ret_val;
int inval;

if( ( ret_val = ioctl( fd48, GPIO_READ_PIN_VAL, &inval ) ) < 0 )
{
    /* ERROR */
}
```

The second uses the **read** system call:

```
int ret_val;
int inval;

if( ( ret_val = read( fd48, (char *)&inval, sizeof( char ) ) ) != sizeof(
char ) )
{
    /* ERROR */
}
```

8.2.4.5. Set GPIO outputs

There are two ways to write GPIOs working as outputs.

The first uses the **IOCTL GPIO_WRITE_PIN_VAL**. The address of the integer variable containing the output value is passed as an argument.

```
int ret_val;

/* Set the output to 1 */
int outval = 1;
if( ( ret_val = ioctl( fd53, GPIO_WRITE_PIN_VAL, &outval ) ) < 0 )
{
    /* ERROR */
}
```

The second uses **write** system call:

```
int ret_val;

/* Set the output to 1 */
int outval = 1;
if( ( ret_val = write( fd53, (char *)&outval, sizeof( char ) ) ) != sizeof(
char ) )
{
    /* ERROR */
}
```

8.2.4.6. Close the device nodes

When done working with a GPIO, its descriptor must be closed to free the resources:

```
/* Close a GPIO descriptor */
close( fd48 );
```

8.2.4.7. How the test application exercises GPIOs

The Digi Embedded Linux environment includes a test application of the GPIOs which uses a button and a LED of the development board. This sample application can be included, as shown in topic 4.2, by selecting the **gpio_test** template.

The source files of the test application are copied to the project folder in **build/apps/gpio_test_c/**

The test application configures **GPIO49** as output, connected to **LED2** on the development board, and **GPIO69** as input, connected to **BUTTON2** on the development board.

Before launching the application, device nodes must be created for these GPIOs, as shown in topic 8.2.3.2.

The test application inverts the LED's value (output) with each press of the button (input). This is done 10 times using IOCTLs, polling the input, 10 times using standard read/write calls, also polling the input, and 10 times with the input configured as IRQ (interrupt mode).

8.2.5. GPIOs on the development board

The development board contains two pushbuttons and two LEDs for test purposes, such as the GPIO test. The buttons and LEDS have these GPIO numbers:

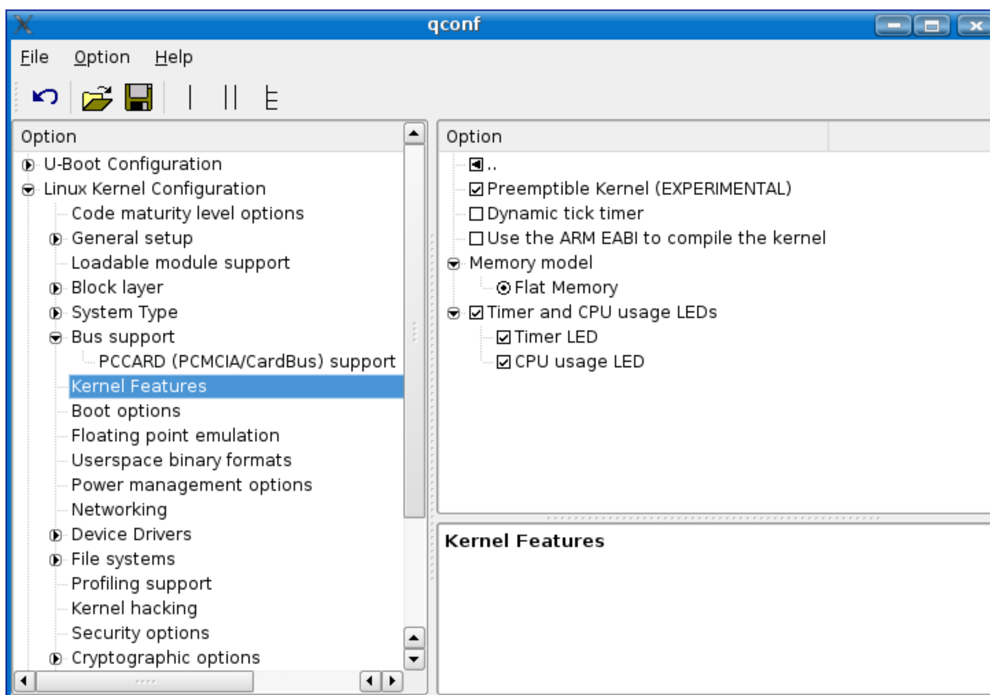
Name	GPIO
BUTTON1	GPIO72
BUTTON2	GPIO69
LED1	GPIO48
LED2	GPIO49

A small kernel driver for giving the LEDs a special usage is provided in the Digi Embedded Linux software.

LED1 can be used as a timer that blinks with a frequency of 1 Hz. This LED can be used, for example, as a visual check that the system is alive and running.

LED2 can be used as a CPU usage meter, which blinks more frequently as the CPU load increases.

These two functions can be enabled in the kernel configuration. Run the configuration tool (described in topic 5.1) Go to **Linux Kernel Configuration > Kernel Features** and activate the **Timer and CPU usage LED** element. Then, check also the other two subelements: **Timer LED** and **CPU usage LED**.



Activating these uses of the development board's LEDs conflicts with the test application of the GPIOs, which uses the same LEDs.

8.3. Ethernet interface

The NS9XX0 processor contains a high performance 10/100 Ethernet controller. This interface to the ConnectCore 9C/Wi-9C and ConnectCore 9P modules is an RJ45 network connector.

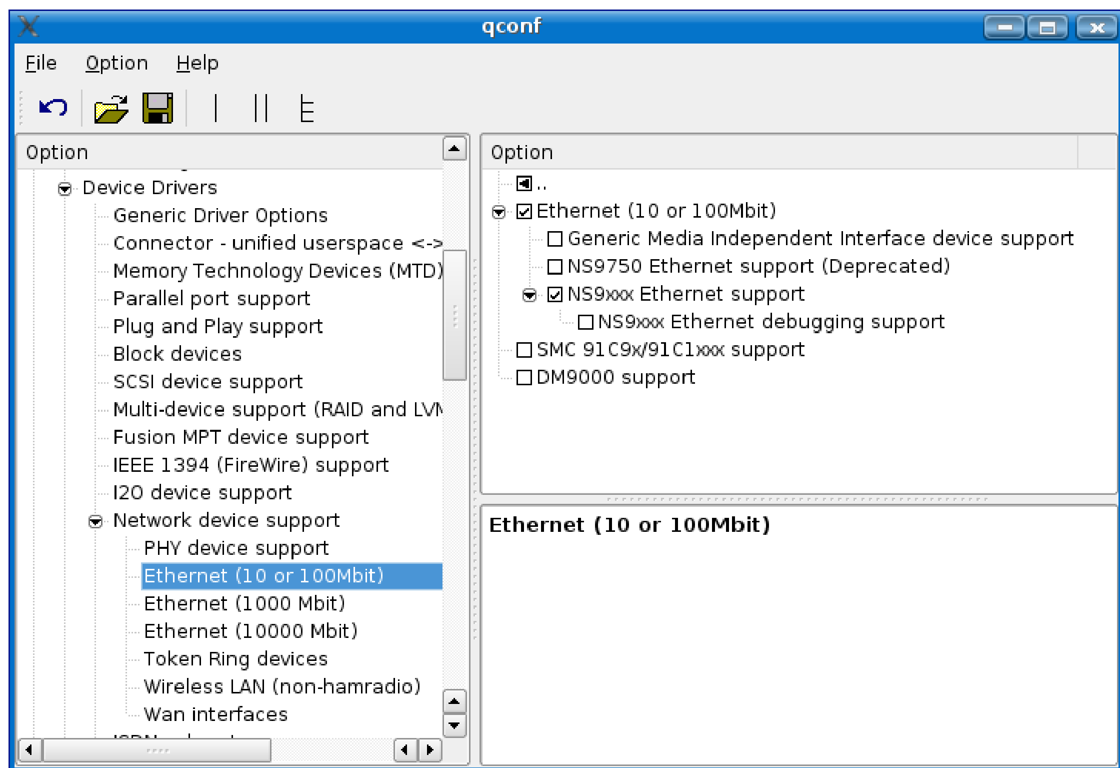
8.3.1. Hardware resources used by the driver

Interface	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
Ethernet	ns9xxx_eth	4,5	50-64			

8.3.2. Enable the Ethernet interface in the kernel

The Ethernet interface is enabled and statically linked into the kernel by default as it is fundamental for any communication with the host system. All configuration settings like IP address and netmask are given to it via the kernel command line interface.

To see where it is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > Network device support > Ethernet (10 or 100Mbit)**.



The Ethernet driver support is included by checking the **NS9xxx Ethernet support** component.

8.3.3. The Ethernet interface in the Linux system

In the Linux system, the Ethernet interface is known as **eth0**. The network settings, which are configured in U-Boot by environment variables (IP address, gateway, netmask, etc.), are automatically assigned to the Ethernet interface after boot.

To enable the Ethernet interface in Linux, enter this command:

```
#   ifconfig eth0 up
```

To disable with the Ethernet interface, enter this command:

```
#   ifconfig eth0 down
```

To assign one or more IP addresses to the Ethernet interface, enter:

```
#   ifconfig eth0 192.168.42.31
#   ifconfig eth0:2 192.168.42.32
#   ifconfig eth0:3 192.168.42.33
```

8.4. Wireless network interface

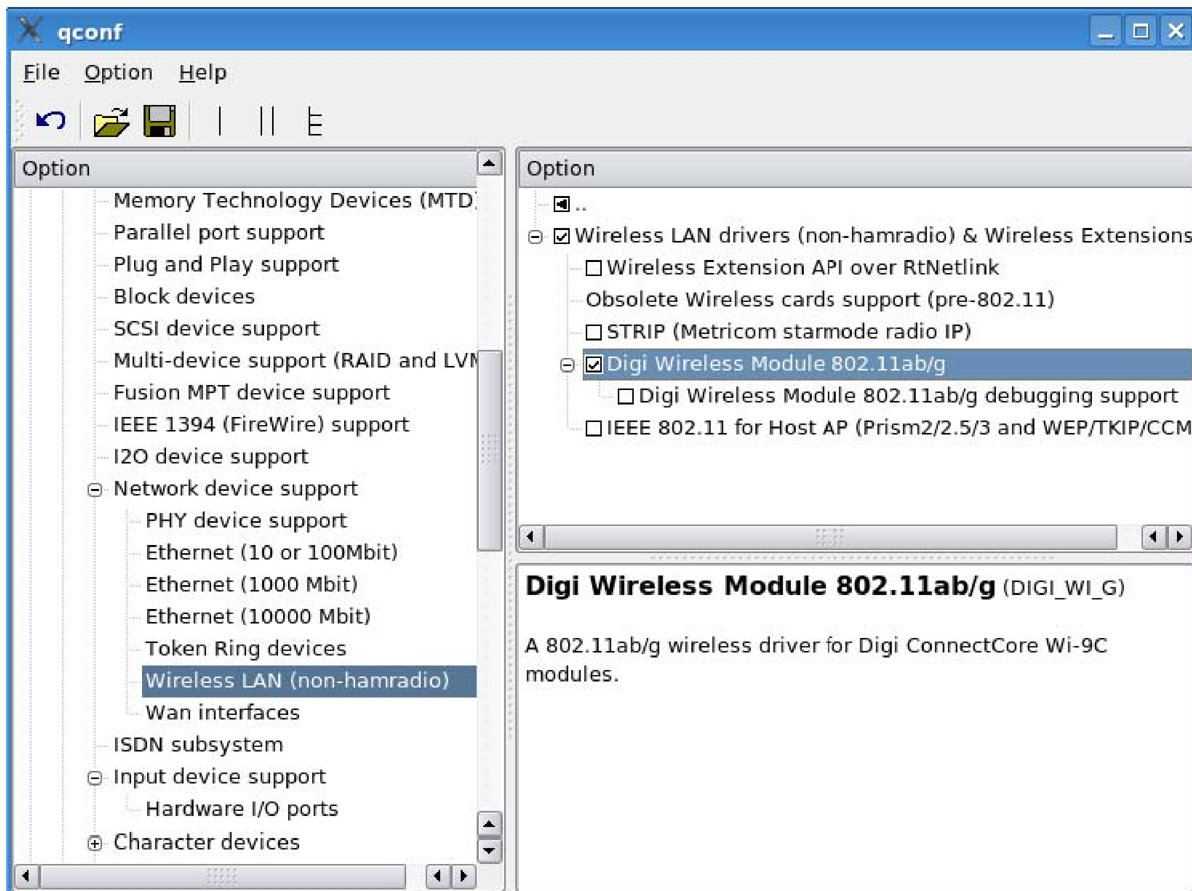
The ConnectCore Wi-9C module contains a Field-Programmable Gate Array (FPGA) which implements an IEEE 802. 11ab/g-compatible wireless network interface. Extensive information about the WLAN adapter is given in topic 9.

8.4.1. Hardware resources used by the driver

Interface	Driver	IRQ	GPIO	Physical memory	Timer	Chip Select
WLAN	digi_wi_g	65	58,65-67	0x6xxxxxxx		static 2

8.4.2. Enable the wireless network interface in the kernel

For ConnectCore Wi-9C platforms, this wireless network interface is enabled and included in the kernel by default. To see where the wireless network interface is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > Network device support > Wireless LAN (non-hamradio)**.



The wireless driver support is included by selecting the **Digi Wireless Module 802.11 ab/g** component.

8.4.3. The wireless interface in the Linux system

In the Linux system, the wireless interface is known as **wlan0**. The network settings, which are configured in U-Boot for the wireless interface by environment variables (IP address, gateway, netmask, etc.), are automatically assigned to the wireless interface after boot.

In Linux, there are also some tools to configure the behavior, encryption and authentication modes of the wireless LAN (WLAN) adapter. For a complete guide in using the WLAN, please consult topic 9.

8.5. Flash memory device

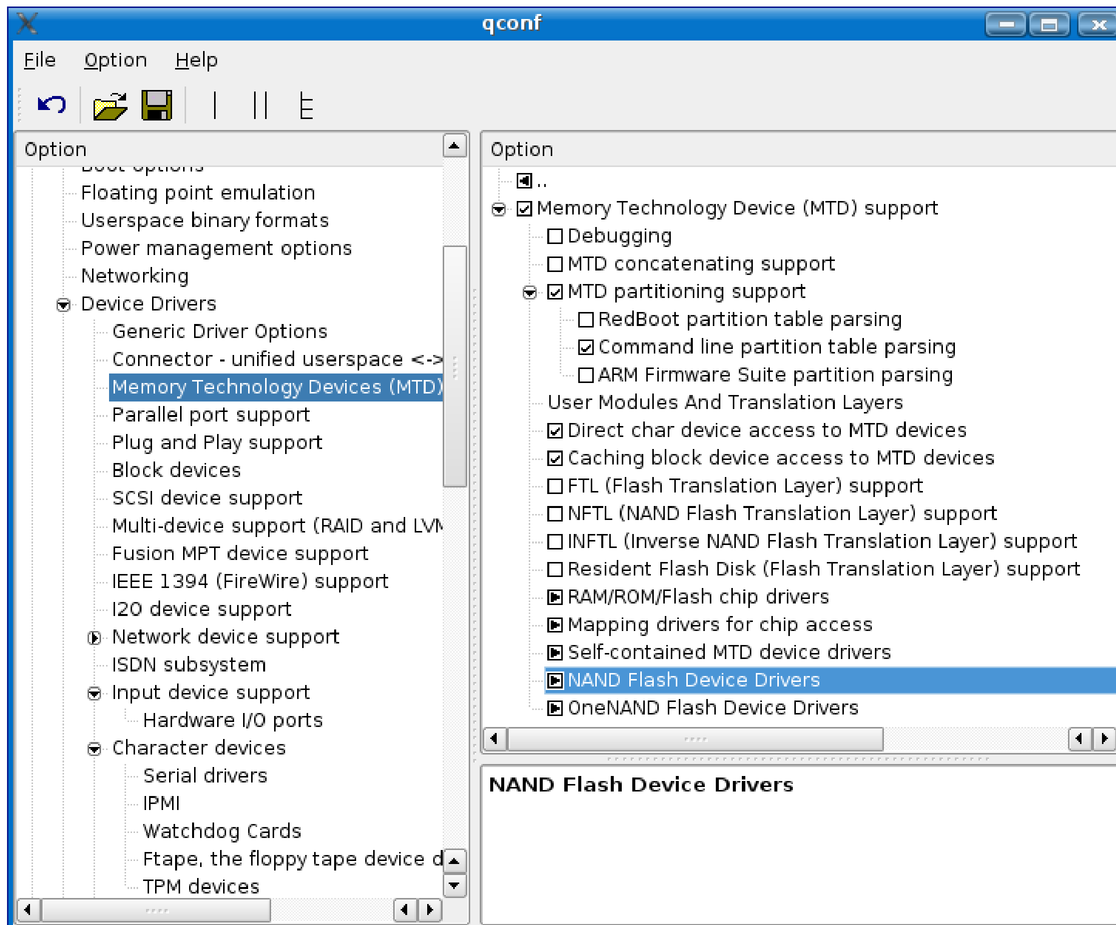
The ConnectCore 9C/Wi-9C and ConnectCore 9P modules contain a Flash memory device for permanent storage of user data, the boot loader, the kernel, the root file system, the wireless FPGA program and the persistent NVRAM settings. The memory chip is a NAND Flash chip which is offered in different size configurations. A Linux driver is used to read from and write to the Flash.

8.5.1. Hardware resources used by the driver

Device	Driver	IRQ	GPIO	Physical memory	Timer	Chip Select
Flash	ccx9x_nand			0x50xxxxxx		static 1

8.5.2. Enable the Flash memory device in the kernel

The Flash memory device is enabled and statically included into the kernel by default. The kernel checks the device on start-up for bad blocks. To see where the Flash memory device is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > Memory Technology Devices (MTD)**.



To support the Flash memory device in Linux, include support for Memory Technology Devices (MTD) and partitioning. The specific driver is included in the **NAND Flash Device Drivers**. To access this area, double-click this item. Check the **NAND Device Support** and **Verify NAND page writes** elements. Then check the specific driver **NAND Flash on ConnectCore 9C/Wi-9C**.

8.5.3. The Flash memory device in the Linux system

In the Linux system, Flash partitions are identified as block devices named **/dev/mtdblock*X***. For example, the default partition structure of the Flash (see topic 7.4.1) is displayed as:

```
#  ls /dev/mtdblock* -l
brw-rw----    1 root     root      31,   0 Jan  1  1970 /dev/mtdblock0
brw-rw----    1 root     root      31,   1 Jan  1  1970 /dev/mtdblock1
brw-rw----    1 root     root      31,   2 Jan  1  1970 /dev/mtdblock2
brw-rw----    1 root     root      31,   3 Jan  1  1970 /dev/mtdblock3
brw-rw----    1 root     root      31,   4 Jan  1  1970 /dev/mtdblock4
brw-rw----    1 root     root      31,   5 Jan  1  1970 /dev/mtdblock5
#  cat /proc/mtd
dev:    size   erasesize  name
mtd0: 000c0000 00004000 "U-Boot"
mtd1: 00040000 00004000 "NVRAM"
mtd2: 00100000 00004000 "FPGA"
mtd3: 00300000 00004000 "Kernel"
mtd4: 01000000 00004000 "RootFS-JFFS2"
mtd5: 02b00000 00004000 "User-JFFS2"
```

Like any other block device, those partitions containing a file system image, such as the rootfs partition, can be mounted with the Linux **mount** command. Then, use normal Linux commands to write to the mounted Flash partition.

Whenever something critical is written to Flash, the system should be synchronized, using the **sync** *command, so that the data is not kept in the cache memory.*

To update images in the partitions, use the **update_flash** application, explained in topic 7.4.2.

Partitions with a read_only flag set in the NVRAM partitions table cannot be updated with update_flash (all partitions are writable by default).

8.6. Serial device driver

The NS9XX0 microprocessor contains four serial ports that can operate in UART or SPI master/slave modes. A serial driver controls the internal serial ports.

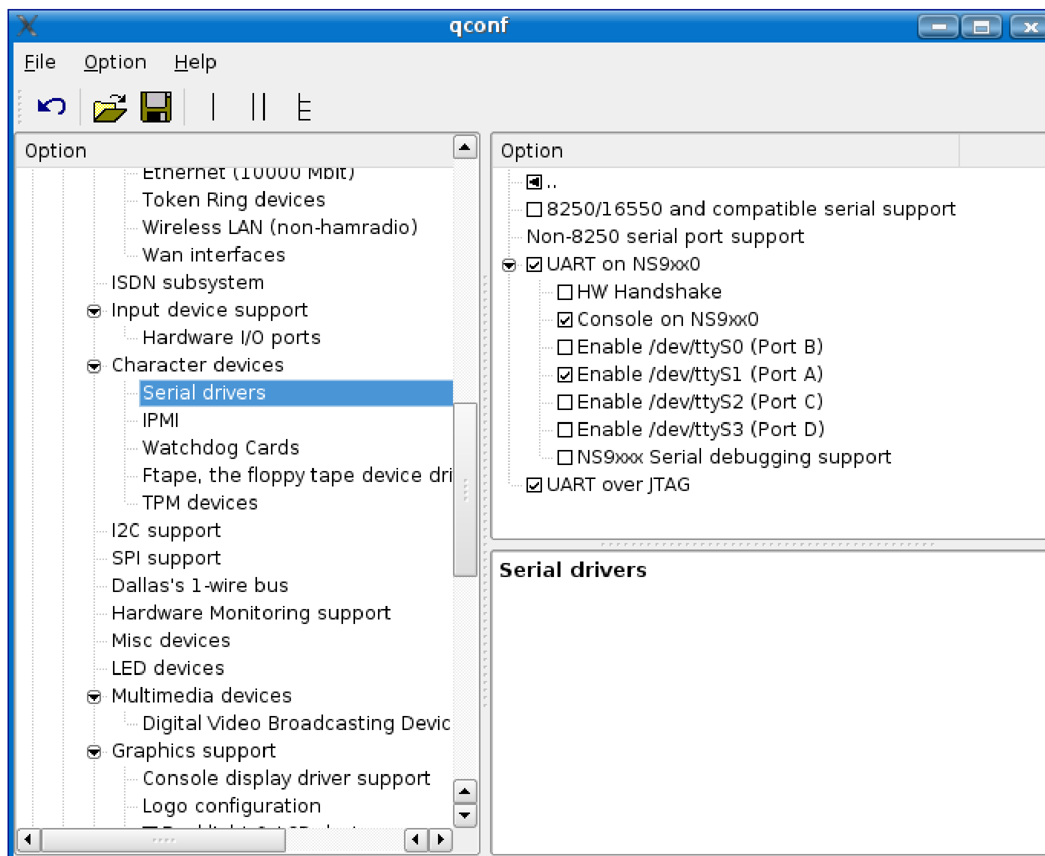
8.6.1. Hardware resources used by the driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
Serial A (UART) /dev/ttyS1	ns9xxx_serial	36,37	8,9 (10-15)[1]			
Serial B (UART) /dev/ttyS0	ns9xxx_serial	34,35	0,1 (2-7)[1]			
Serial C (UART) /dev/ttyS2	ns9xxx_serial	38,39	40,41 (42,43, 20-23)[1]			
Serial D (UART) /dev/ttyS3	ns9xxx_serial	40,41	44,45 (46,47, 24-27)[1]			

[1] Only when HW Handshaking is enabled

8.6.2. Enable the serial device driver in the kernel

By default, only Serial Port A is enabled in UART mode. To see which serial ports are enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > Character devices > Serial drivers**.



Enabling the element **UART on NS9xx0** grants access to enabling each of the four serial ports in UART mode.

To enable the console in a serial port, check **Console on NS9xx0**. The serial port connected to the console is selected with the **console** environment variable in U-Boot. Refer to topic 7.2 for information about how to change the value of U-Boot variables.

Enabling the option **HW Handshake** will enable the hardware handshake on the serial ports enabled in UART mode.

Before enabling UART mode for a serial port, make sure that the SPI mode is not enabled for the same port (see topic 8.7.2).

The development board contains switches for configuring the signals of the serial ports. For more information, see the **ConnectCore 9C/Wi-9C Hardware Reference manual** *or the* **ConnectCore 9P Hardware Reference** *.*

8.6.3. Identify serial devices in the system

In the Linux systems, serial ports working in UART mode appear as character devices named **/dev/ttyS*X***. For example, in the factory default image, the **ls /dev/ttyS*** command displays this serial device:

```
#   ls /dev/ttyS*
/dev/ttyS1
```

If enabled, ttyS0 is Serial Port B and ttyS1 is Serial Port A.

8.6.4. Manage serial ports from the user space

The standard serial programming API applies to the NS9xx0 serial ports. For information about serial programming, see the *Serial Programming HOWTO* at **http://tldp.org/HOWTO/Serial-Programming-HOWTO/index.html** or the *Serial Programming Guide for POSIX Operating Systems* at **http://www.easysw.com/~mike/serial/serial.html**.

8.7. Serial Peripheral Interface (SPI) mode

The four serial ports of the NS9XX0 processor can be configured in SPI mode (master or slave). By default, no serial port is configured in SPI mode.

8.7.1. Hardware resources used by the SPI driver

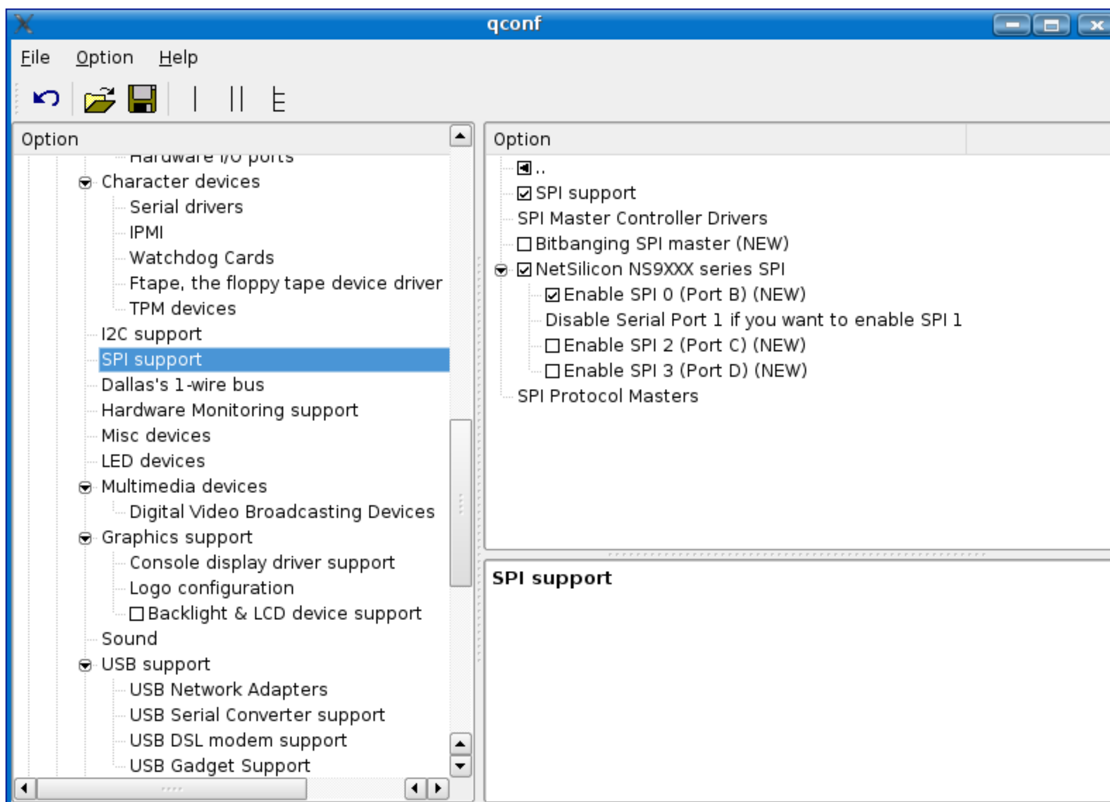
Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
Serial A (SPI mode)	ns9xxx_spi	60,61	8,9,14,15			
Serial B (SPI mode)	ns9xxx_spi	58,59	0,1,6,7			
Serial C (SPI mode)	ns9xxx_spi	62,63	40,41,22,23			
Serial D (SPI mode)	ns9xxx_spi	64,65	44,45,26,27			

8.7.2. Enable the SPI device in the kernel

Since all serial ports are configured in UART mode, SPI support is disabled by default.

Before enabling SPI mode for a serial port, make sure that the UART mode is not enabled for the same port (see topic 8.6.2).

To enable SPI support, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > SPI support**. Enable **SPI support** and **NetSilicon NS9XXX series SPI** elements as built-in kernel features. That grants access to enabling each of the four serial ports in SPI mode, except for those which are enabled in UART mode.



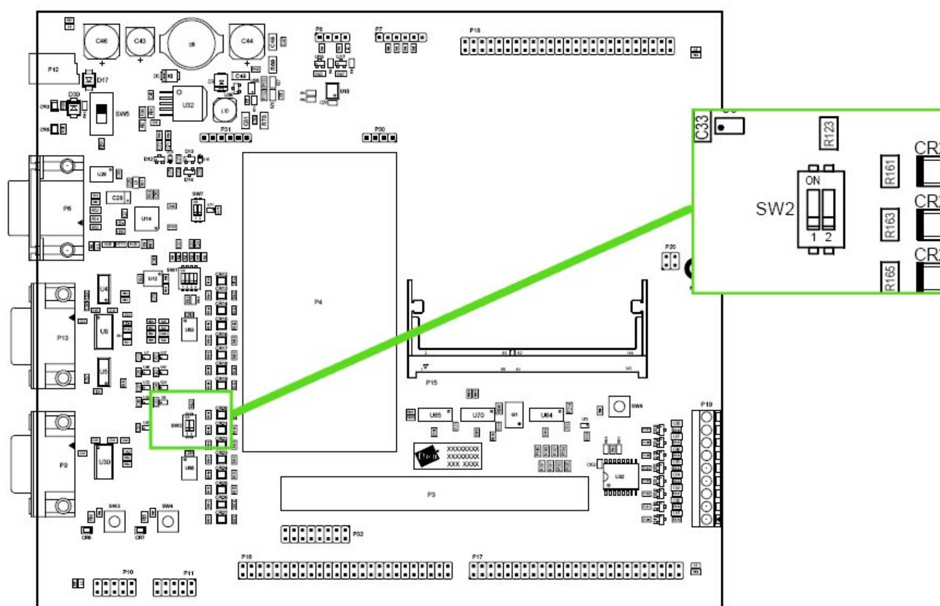
8.7.3. Access to SPI bus

The SPI bus cannot be accessed directly from user space. Instead, it is access via the SPI client drivers, like the Touch screen driver.

8.8. Touch screen

If an LCD Application Kit was purchased, the provided TFT LCD contains a Touch Screen sensor and an SPI touch screen controller (ADS 7846). The touch screen lines come together with the LCD lines in a single cable. Internally, the touch screen lines are connected to Serial Port B.

To use Serial Port B as a Serial Peripheral Interface (SPI), turn off microswitch **SW2.2** on the development board.



SW2	ON	OFF
SW2.2	UART mode	SPI mode

A Linux driver implements the support for the ADS 7846 touch screen controller.

8.8.1. Hardware resources used by the driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
Touch screen ADS7846	ads7846	EXT 3	0,1,6,7			

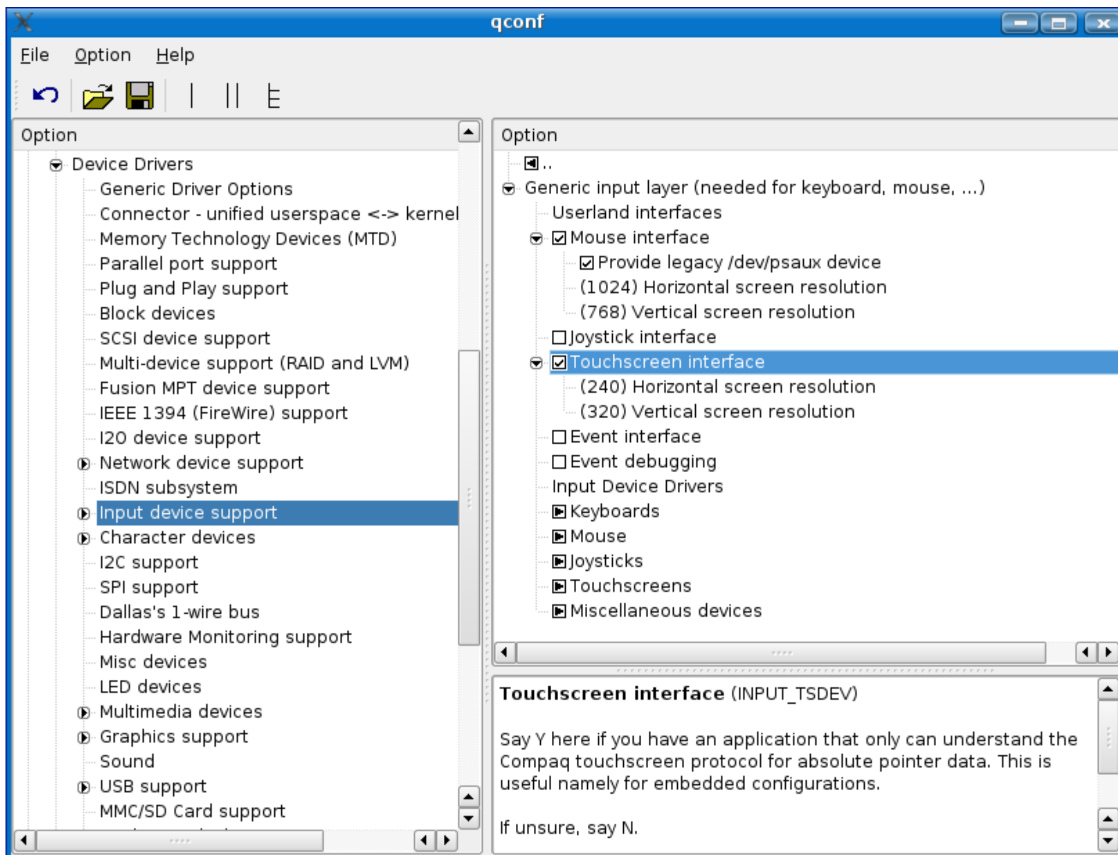
8.8.2. Enable the touch screen device in the kernel

The touch screen device is not included in the default kernel configuration.

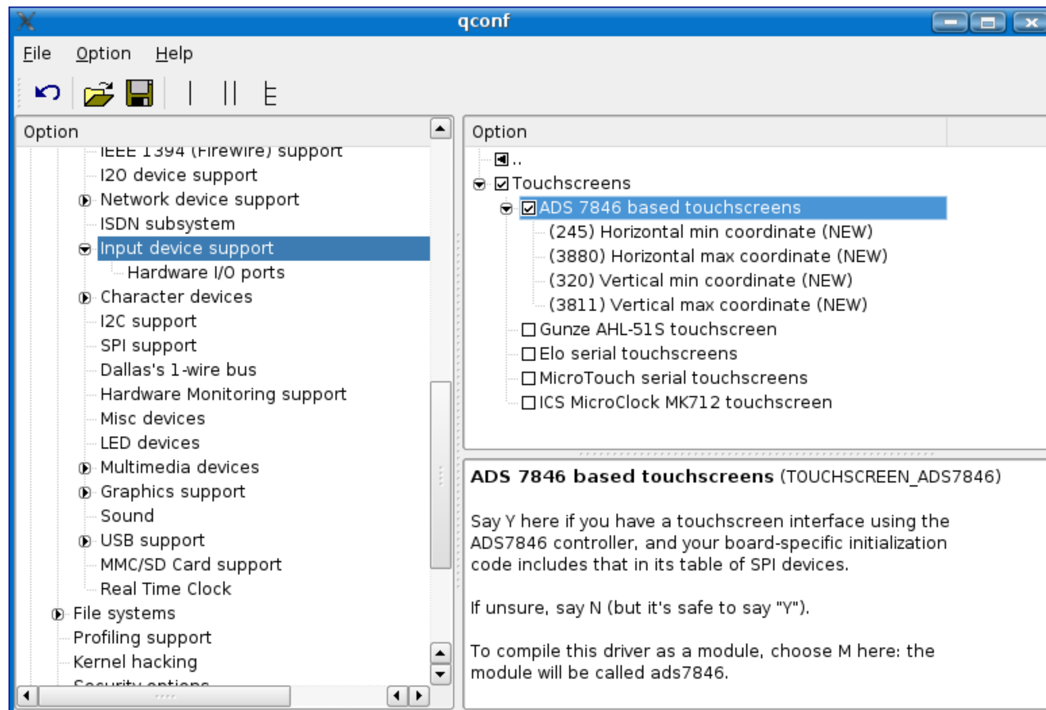
Enable the touch screen device for a TFT LCD with a touch screen based in the ADS 7846 controller only.

To work with the touch screen driver, make sure that Serial Port B is not enabled in UART mode (see topic 8.6.2) and turn off microswitch **SW2.2** on the development board.

To enable the touch screen driver, open the configuration tool and select **Linux Kernel Configuration > Device Drivers > Input device support**.



Enable the element **Touchscreen interface**. Then click the **Touchscreens** element and enable **ADS 7846 based touchscreens**.



The coordinates of the touch screen limits are approximate. Different values may needed to be configured to calibrate the touch screen properly.

8.8.3. Identifying the touch screen device in the Linux system

In the Linux system, the touch screen appears as devices named **/dev/ts0**.

8.8.4. Manage the touch screen device from user space

The touch screen is hardly ever accessed directly. Graphics libraries, like the ones listed in topic 11, take care of managing the touch screen device.

8.9. USB host interface

The NS9xx0 processor contains a USB 2.0 host interface that supports full-speed (12 Mbps) and low-speed (1.5 Mbps). The interface is extracted to the ConnectCore 9C/Wi-9C and ConnectCore 9P module in the form of two USB host ports.

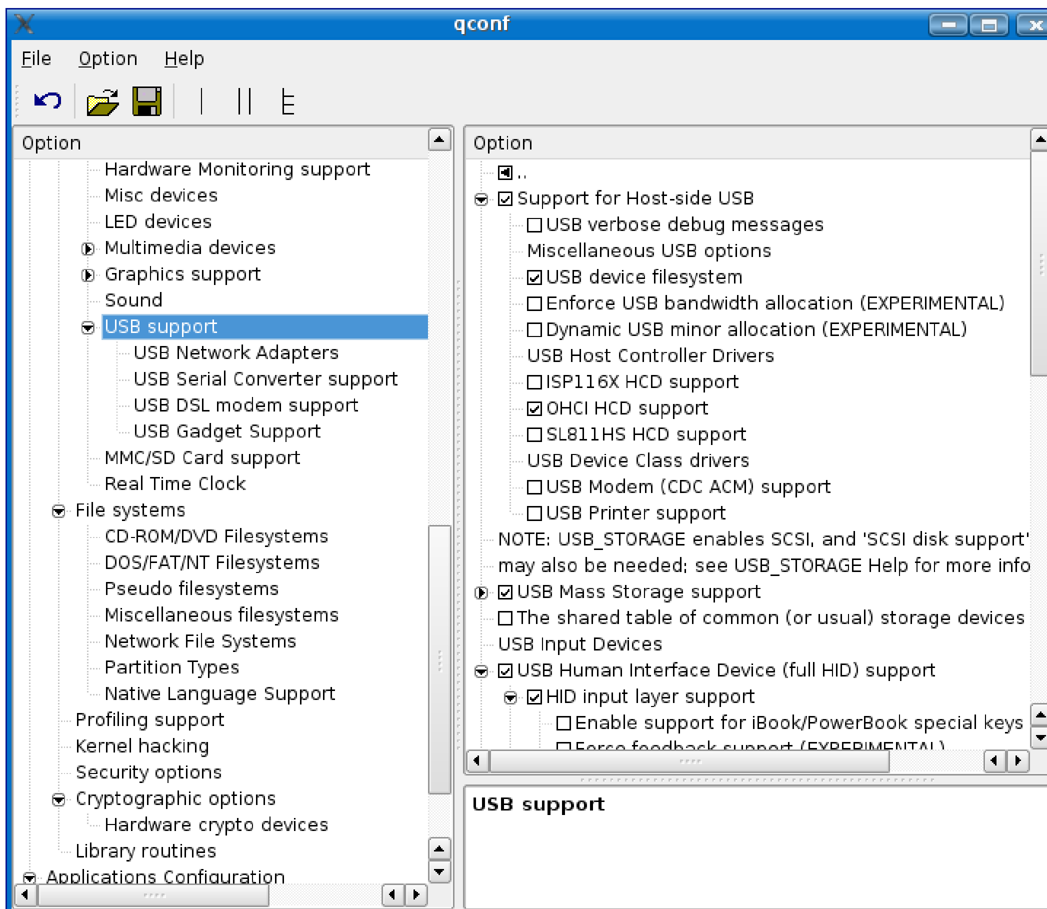
8.9.1. Hardware resources used by the USB host interface driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
USB Host		25				

8.9.2. Enable the USB host interface in the kernel

By default, the USB host interface is enabled in the kernel.

To see where the USB host interface is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > USB support**.



Check whether the items shown in the figure have USB support for storage media, such as USB Flash disks, and for HID input devices, such as a USB mouse.

8.9.3. Identify the USB devices in the Linux system

USB devices are identified as soon as they are plugged in. Depending on the log level, a message reporting that the USB device that has been plugged in may be displayed on the serial console. For example, plugging in a USB memory stick displays this message:

```
Vendor: 32MB      Model: HardDrive        Rev: 1.88
   Type:   Direct-Access                     ANSI SCSI revision: 02
SCSI device sda: 64000 512-byte hdwr sectors (33 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
SCSI device sda: 64000 512-byte hdwr sectors (33 MB)
sda: Write Protect is off
sda: assuming drive cache: write through
sd 1:0:0:0: Attached scsi removable disk sda
```

If nothing is displayed to the console, view all the system messages by printing the contents of the system log file:

```
#   tail /var/log/messages
```

8.9.4. Manage the USB interface from user space

8.9.4.1. USB Memory sticks

USB memory sticks must be mounted before using them. To mount a USB memory stick, use **lxmount**. To umount a USB memory stick, use **lxumount**.

```
#   lxmount
#   ls /media/
sda1
#   lxumount
#   ls /media/
#
```

8.9.4.2. Other USB input devices

Other USB input devices, such as a mouse or a keyboard can be used immediately after being connected.

For example, connect a keyboard to a USB connector. Then, in the serial console, run:

```
#   cat /dev/tty0
```

Now, on the keyboard type some text, ending with **CTRL+D**. The typed text will be displayed in the serial console.

A USB keyboard in combination with an LCD or VGA monitor can also be used for the console. To mount a USB memory stick, use, change the **std_bootarg** variable in U-Boot:

```
#   setenv std_bootarg console=ttyS1,38400 console=tty0
#   saveenv
```

8.10. I²C

The NS9XX0 processor contains an I²C v.1.0 port, which can be configured in both master and slave modes. A custom driver has been developed for this interface.

Additionally, the development board contains an I²C 8-bit I/O device (Philips PCA9554). A driver for this I/O port is also provided.

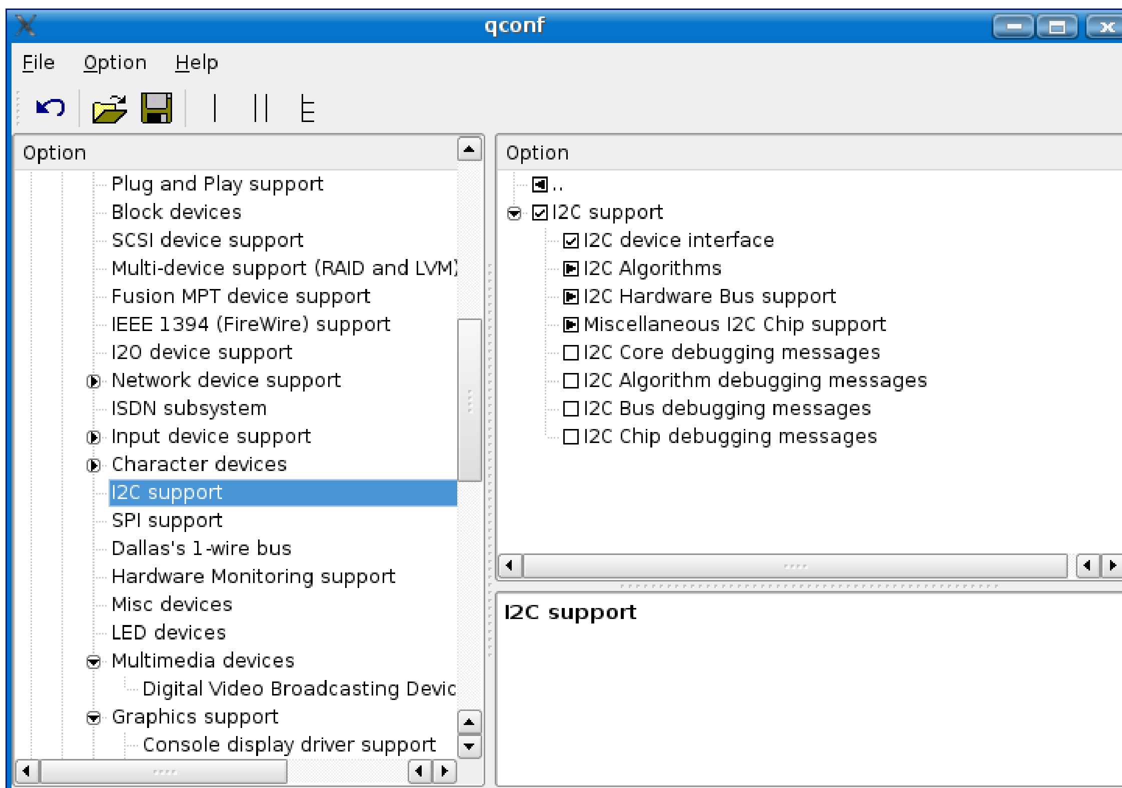
8.10.1. Hardware resources used by the I²C interface

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
I²C	i2c-ns9xxx		46,47			
I²C I/O port PCA9554	pca9554					

8.10.2. Enable the I²C interface in the kernel

By default, the I²C interface is enabled and built-in into the kernel.

To see where the I²C interface is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > I2C support**.



In the displayed configuration information, the following items must be checked for having support for the I^2C interface and the I^2C I/O port:

- **I2C support**
  - **I2C device interface**
  - **I2C Algorithms**
    - **I2C bit-banging interfaces**
  - **I2C Hardware Bus support**
    - **NS9360/NS9750 I2C bus**
  - **Miscellaneous I2C Chip support**
    - **Philips PCA9554(A) 8-bit I/O port**

8.10.3.Identify the I^2C interface in the Linux system

The I^2C bus is populated in the Linux system within the **/sys** folder of the rootfs. A character device, **/dev/i2c-0**, is also created for accessing the bus in raw mode.

The PCA9554 I2C I/O port is populated in the system with several file descriptors in the **/sys** folder:

```
#   ls -la /sys/devices/platform/i2c-0/0-0020
drwxr-xr-x    2 root     root            0 Jan  1 00:47 .
drwxr-xr-x    3 root     root            0 Jan  1  1970 ..
lrwxrwxrwx    1 root     root            0 Jan  1 00:47 bus ->
../../../../bus/i2c
-rw-r--r--    1 root     root         4096 Jan  1 00:47 config
lrwxrwxrwx    1 root     root            0 Jan  1 00:47 driver ->
../../../../bus/i2c/drivers/pca9554(a)
-r--r--r--    1 root     root         4096 Jan  1 00:47 input
-r--r--r--    1 root     root         4096 Jan  1 00:47 name
-rw-r--r--    1 root     root         4096 Jan  1 00:47 output
-rw-r--r--    1 root     root         4096 Jan  1 00:47 polinv
--w-------    1 root     root         4096 Jan  1 00:47 uevent
```

8.10.4. Manage the I^{2}C interface from user space

Normally, I^{2}C devices, such as the PCA9554 I/O port, are accessed from applications using the API provided by the class driver. However the I^{2}C bus can be accessed directly using the device **/dev/i2c-0**.

For more information about writing directly to the I^{2}C bus, consult the Linux documentation at:

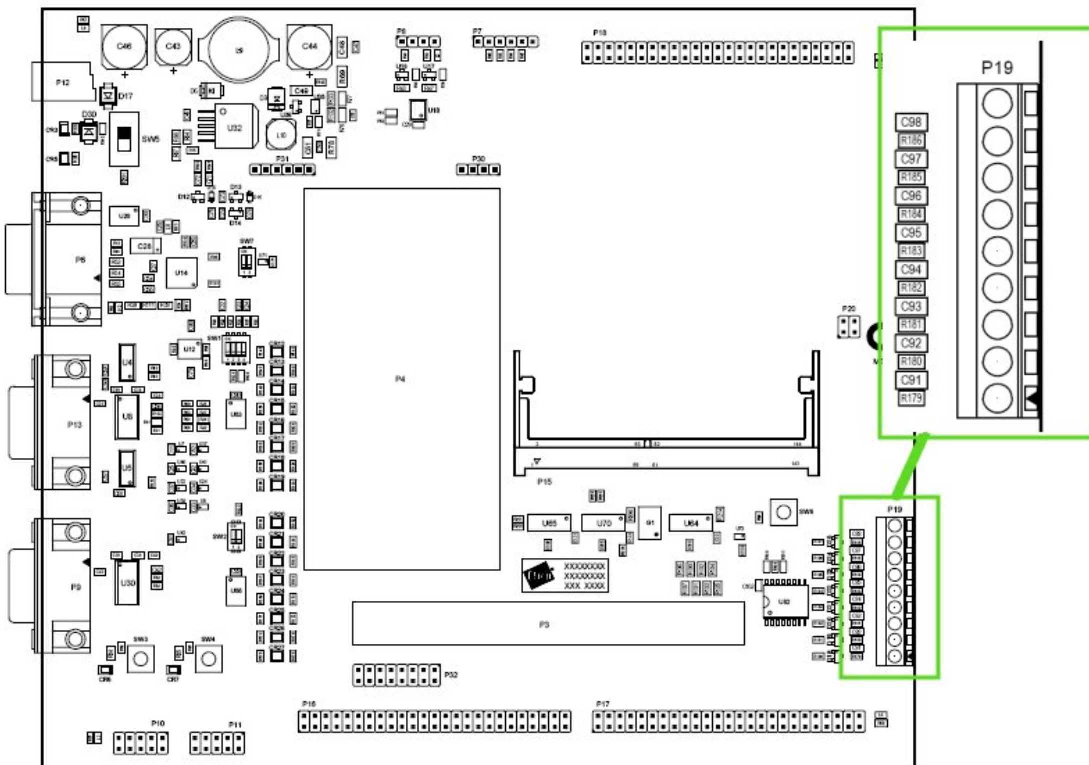
/usr/local/DigiEL-4.0/kernel/linux/Documentation/i2c/dev-interface

8.10.4.1. PCA9554 8-bit I/O port

The PCA9554 I/O port can be accessed via the file descriptors **config**, **input**, **output** and **polinv** available at **/sys/devices/platform/i2c-0/0-0020/**.

The **config** descriptor configures the I/Os as inputs or outputs. **input** is used to read the inputs values. **output** is used to write the outputs values. **polinv** is used to invert the polarity of the I/Os.

A sample application is provided in folder **/usr/local/DigiEL-4.0/apps/i2c_gpio_test/**. This application checks for possible shortcuts between adjacent I/Os. Running this application, requires interconnecting I/Os 0-2-4-6 and 1-3-5-7 respectively, which can be found in connector P19 on the development board:



8.11. Real Time Clock (RTC)

The NS9360 processor contains a real time clock (RTC) module that tracks the time of the day to an accuracy of 10 milliseconds and provides calendar functionality that tracks day, month, and year. A character driver for the RTC is provided.

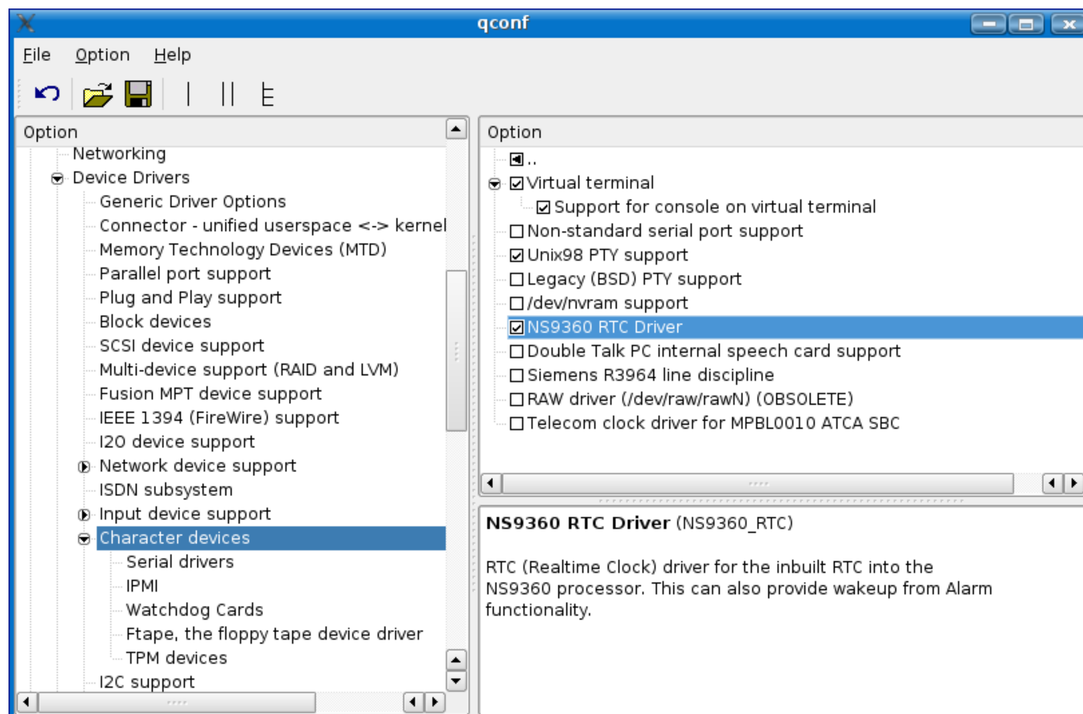
Since the RTC is internal to the processor, no battery maintains the date/time. This RTC device is mainly intended for triggering alarms and scheduled jobs.

8.11.1. Hardware resources used by the RTC driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
RTC	ns9360_rtc					

8.11.2. Enable the RTC device in the kernel

By default, support for the NS9360 RTC device is enabled and included into the kernel. To see where the RTC device is enabled, open the configuration tool and go to **Linux Kernel Configuration > Device Drivers > Character devices**. The **NS9360 RTC Driver** should be selected.



8.11.3. Identify the RTC device in the Linux system

The device driver is accessible via the file descriptor **/dev/rtc**.

8.11.4. Manage the RTC device from user space

The RTC uses the Linux standard API for RTC devices. For more information, see the Linux documentation at **/usr/local/DigiEL-4.0/kernel/linux/Documentation/rtc.txt**. A sample application

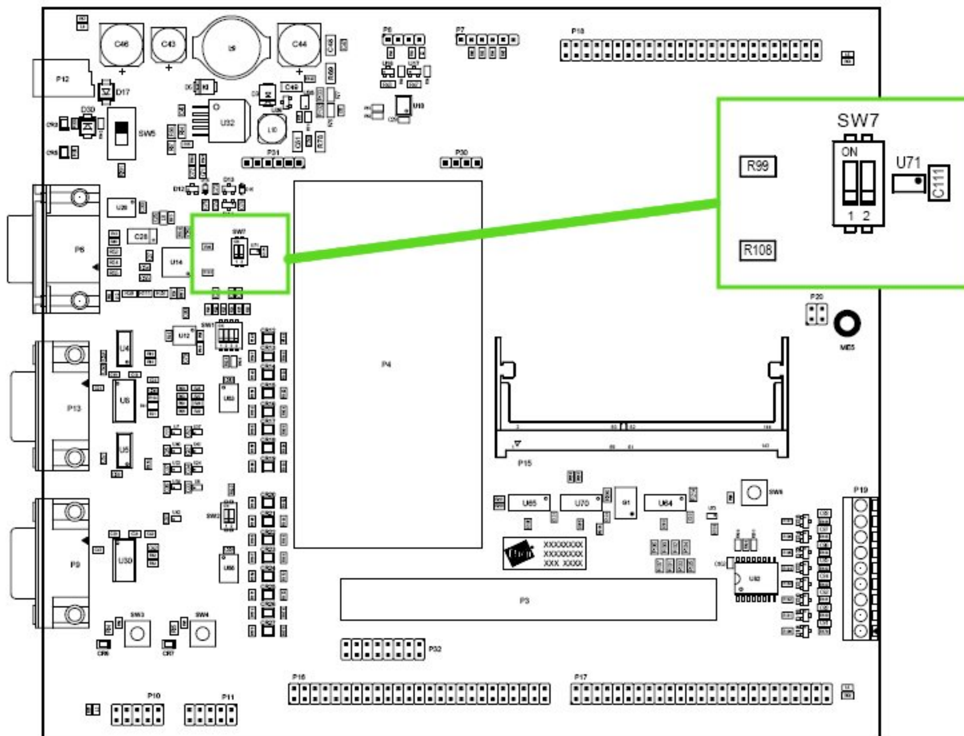
is in **/usr/local/DigiEL-4.0/apps/rtc_test_c/**. This sample application performs several operations on the RTC including read/set the current time, set the alarm, test the periodic interrupt, etc.

8.12. Video/Graphics support

There are two options for video/graphics support using the ConnectCore 9C/Wi-9C and ConnectCore 9P modules: TFT LCD display or VGA monitor.

The NS9360 microprocessor contains a flexible LCD controller for TFT LCD displays. The development board contains a VGA DAC (external to the module) that converts the digital signal to analog, for VGA monitors.

Only one video configuration can be used at a time on the development board: either TFT LCD display or VGA monitor. Micro switch SW7 on the development board configures this setting.



SW7	ON	OFF
SW7.1	VGA disabled	VGA enabled
SW7.2	not used	

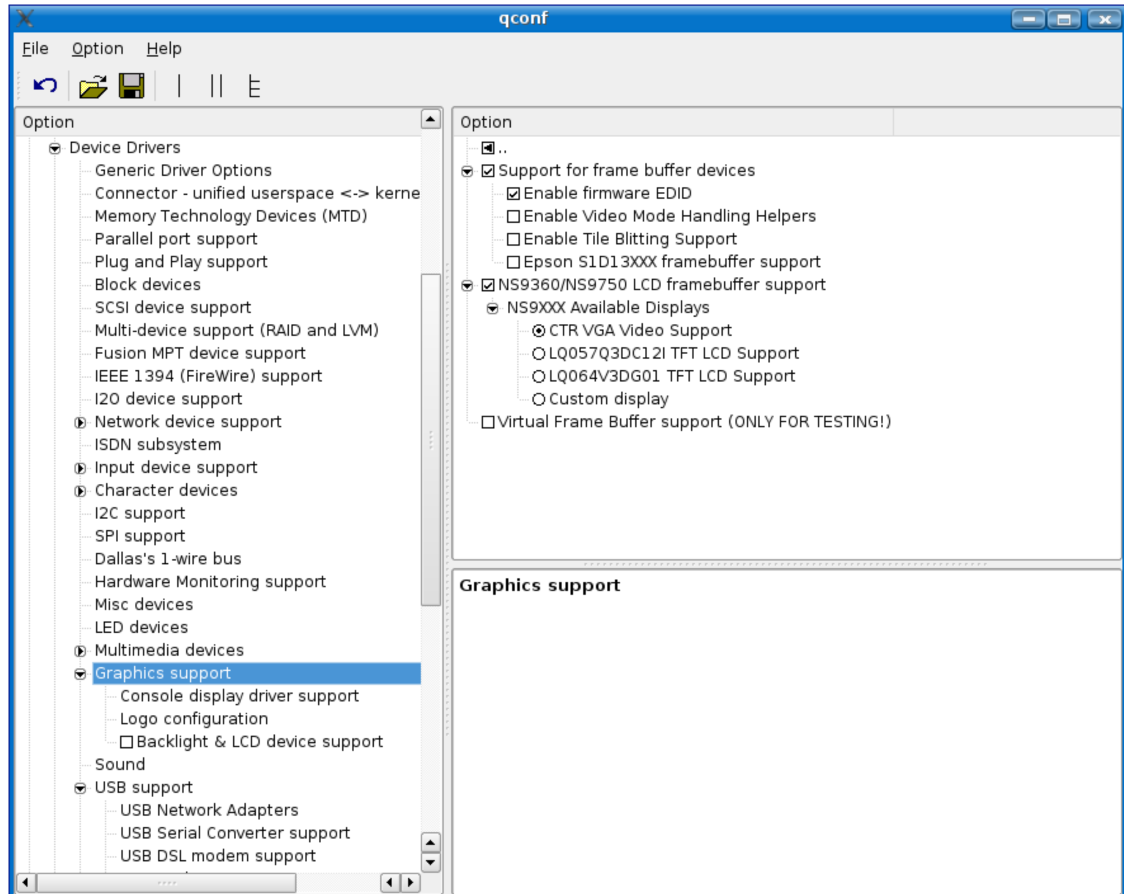
8.12.1. Hardware resources used by the video driver

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
Display	ns9xxxfb		15,18-41			

8.12.2. Including graphics support in the kernel

By default, graphics support is included in the kernel for VGA monitors.

The configuration tool allows removing graphics support, changing it for any of the supported TFT LCD displays, or selecting to compile the graphics driver as module. To examine the graphics support, do a **make xconfig** and go to **Linux Kernel Configuration > Device Drivers > Graphics support**.



To remove graphics support, uncheck the **NS9360/NS7520 LCD framebuffer support.**

To compile graphics support as a module, click until a black circle is displayed in the checkbox for the graphics module named **ns9750fb**.

To include support for a TFT display, select one of the available TFT LCDs. If working with a TFT LCD that does not appear in the list, select **Custom display**.

TFT LCD displays only work if Serial Port B is NOT enabled in UART mode. To disable UART mode of serial ports, see topic 8.6.2

8.12.2.1. Custom graphics display support

To add support for a custom TFT display, that is, TFT LCD that does not appear in the list of supported TFTs, select **Custom display** in the **Graphics support** part of the Linux Kernel Configuration.

Supporting a custom TFT display also requires modifying a kernel header file. See topic 5.9 for details on modifying kernel sources. Modify the header file **linux/drivers/video/displays/ns9*xxx*/custom.h** with the appropriate display settings for:

- Resolution
- Timing values
- GPIO to use for LCD power

8.12.3. Identify the video graphics device in the system

When the kernel loads the video graphics support, it creates a device in the file system called **/dev/fb0**.

If the driver is working, a Linux penguin appears in the display.

8.12.4. Manage the video graphics display from the user space

Normally framebuffer memory is not written to directly. Standard graphics frameworks such as Qtopia or Nano-X can handle the display device and provide a higher API layer for graphical applications.

8.13. High-performance counter

To help develop drivers and measure critical timings, Digi Embedded Linux provides a high-performance counter, named **rdtsc**. This high-performance counter is similar to the **rdtsc** command on x86 processors. The high-performance counter is a timer running with CPU clock frequency, that measures instruction times.

rdtsc is not a driver, but it can be used by other drivers in debug code. For example, the Ethernet driver at **kernel/linux/drivers/net/arm/ns9xxx_eth.c** uses it.

8.13.1. Hardware resources used by the high-performance counter interface

Device	Driver	IRQ	GPIO	Physical Memory	Timer	Chip Select
High-Performance Counter					1	

8.13.2.Use the high-performance counter in the kernel

To use the high-performance counter, the driver must include this code:

```
/* for timing collection */
enum {
        START_XMIT = 0,
        START_XMIT_DONE,
};

#define NAME(x) #x

static const char* ns_hperf_type_names[] = {
        NAME( START_XMIT ),
        NAME( START_XMIT_DONE ),
};

/* ns_hperf_type_names needs to be define first before including */
# include <asm/arch-ns9xxx/ns9xxx_hperf.h>
```

The high-performance counter can then be used at any place in the code with these instructions:

Instruction	Description
ns_hperf_mark(START_XMIT)	Take a timing mark (the first instance initiates the counter).
ns_hperf_mark(START_XMIT_DONE)	Take the last mark (the counter is stopped).
ns_hperf_marks_dump()	Print the timing results.

For example:

```
ns_hperf_mark( START_XMIT );
printk( "A\n" );
ns_hperf_mark( START_XMIT );
printk( "AA\n" );
ns_hperf_mark( START_XMIT_DONE );
ns_hperf_marks_dump();
```

Timings are then printed in the kernel messages. The resolution is presented first, followed by the absolute time, and the delta (+) since the mark before.

The output for the code above is:

```
A
AA
ns_hperf: Providing timing resolution of 6 ns for 27 seconds
Timing information
 START_XMIT              : (start)
 START_XMIT              : 00000048 us (+000048)
 START_XMIT_DONE         : 00000070 us (+000022)
```

9. Use the WLAN adapter

The ConnectCore Wi-9C embedded module includes a Wireless LAN adapter, integrated in the module. This adapter is designed to comply with IEEE 802.11b/g Wireless LAN standard.

This topic explains the details of the WLAN adapter of the ConnectCore Wi-9C, configuration, security, firmware...

If working with a non-wireless module, skip this topic.

Working and communicating with the ConnectCore Wi-9C via the wireless interface requires that a wireless Access Point be installed and configured for the network.

9.1. Wireless security concepts

One of the most important concerns when talking about wireless communications is the security and integrity of the data. For this reason it is necessary to introduce two concepts: encryption and authentication.

Encryption is the process of making data unreadable without a certain deciphering key.

Authentication is the act of confirming the identity or provenance of something or someone.

9.2. Features of the WLAN adapter

Features of the WLAN adapter include:

- Complies with the IEEE 802.11b and IEEE 802.11g 2.4Ghz (DSSS) standards.
- High data transfer rate – up to 54Mbps.
- Supports 64/128-bit WEP, TKIP and AES encryption.
- Supports open, WPA and WPA2 (personal and enterprise) authentication.

For WLAN security issues, the card supports 64/128-bit WEP data encryption that protects the wireless network from eavesdropping. It also supports WPA-PSK (Wi-Fi Protected Access) feature that combines IEEE 802.1x and PSK (Pre-Shared Key), TKIP (Temporal Key Integrity Protocol) technologies. Client users are required to authorize before accessing to APs or AP Routers, and the data transmitted in the network is encrypted/decrypted by a dynamically changed secret key. Furthermore, this adaptor supports WPA2-PSK function, which provides a stronger encryption mechanism through AES (Advanced Encryption Standard), which is a requirement for some corporate and government users. Hardware encryption support is available for AES.

WPA and WPA2 enterprise is also supported by the **wpa_supplicant** package.

9.3. Include the wireless interface in the Linux kernel

The WLAN driver is included in the default kernel for ConnectCore Wi-9C platforms. Topic 8.4.2 explains how to enable the WLAN driver in the kernel.

9.3.1. Necessary components

Several applications are necessary for configuring the WLAN adapter:

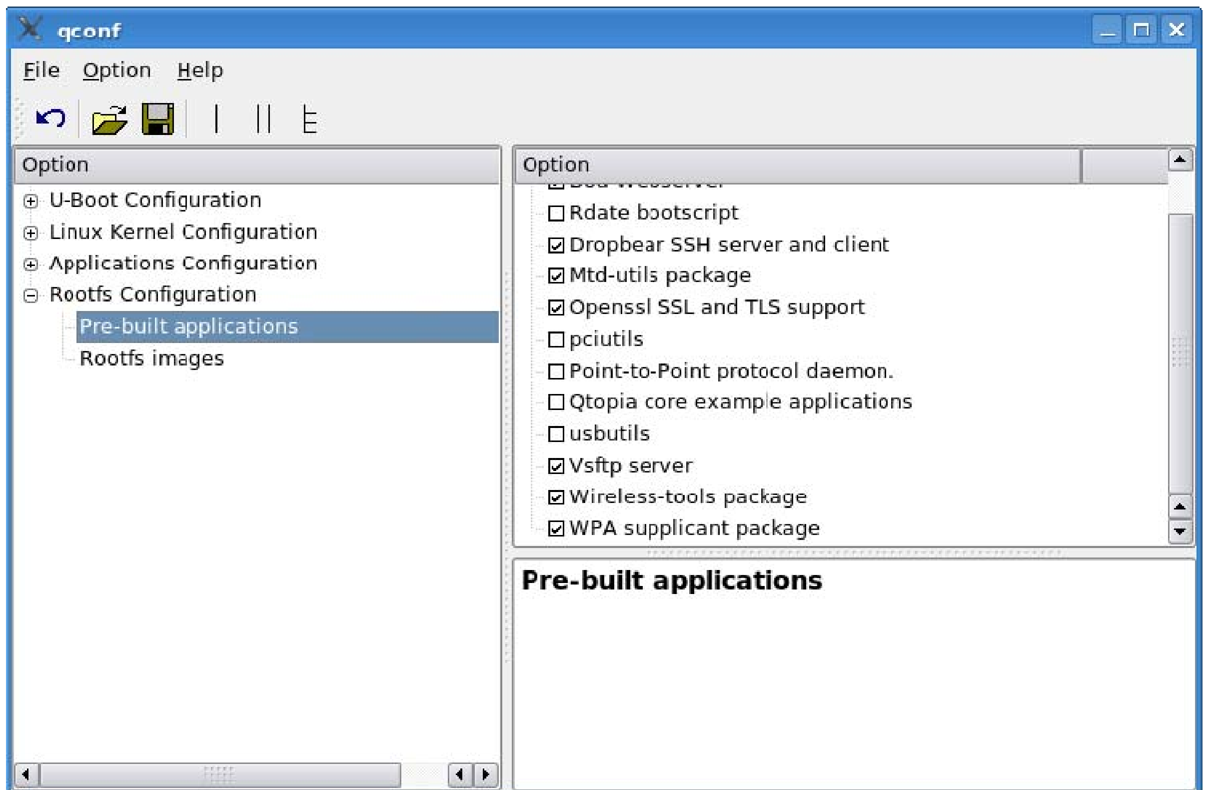
- Wireless-tools package
- WPA supplicant package

The Wireless-tools package is a set of command line tools allowing manipulation and configuration of WLAN adapters.

The WPA supplicant package implements key negotiation with a WPA Authenticator and it controls the roaming and IEEE 802.11 authentication/association of the WLAN driver.

If the project contains rootfs support, these two packages are included by default in the rootfs.

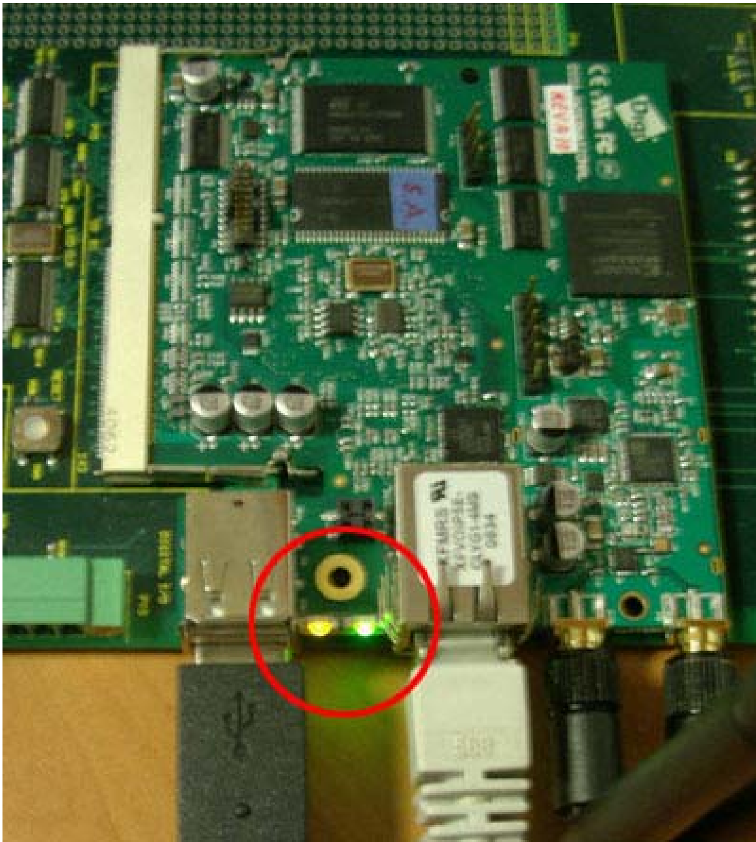
To see where they are enabled, open the configuration tool and go to **Rootfs configuration > Prebuilt applications**.



Both packages should be checked.

9.4. Wireless interface LEDs

ConnectCore Wi-9C module has 2 LEDs (one green, one yellow) between the Ethernet and USB host connectors, related to the wireless interface.



The green LED indicates the wireless status. It can represent several states:

When LED is:	Wireless interface is:
Solid ON	Connected to an access point
Slow blinking	Connected to an ad-hoc computer
Fast blinking	Scanning
Solid OFF	Not Connected

The yellow LED indicates network activity. It blinks when packets are being transmitted.

9.5. Network settings of WLAN interface

The network settings of the WLAN interface are the MAC and IP addresses and the network mask. These three parameters are configured in U-Boot by means of the following environment variables:

Parameter	U-Boot environment variable	Default value
MAC address	wlanaddr	-
IP address	ipaddr_wlan	192.168.43.30
Network mask	netmask_wlan	255.255.255.0

The MAC address of the WLAN interface is unique for each module and it is set during the production process. The MAC address can be seen on a sticker on the module.

9.5.1. Modify network parameters

The IP address and network mask of the WLAN can be modified with the U-Boot **setenv** command, for example:

```
#   setenv ipaddr_wlan 192.168.43.30
#   setenv netmask_wlan 255.255.255.0
#   saveenv
```

Linux applies these settings to the WLAN interface during the boot process.

9.6. Basic wireless operations

For first tests please configure an Access Point with ESSID **default** and disable encryption. The AP should have an IP address in the 192.168.43.0 subnet.

9.6.1. Show the current wireless network status

There are several ways to get information about the WLAN interface.

The **ifconfig** command shows basic information of the wireless device such as IP address, netmask, etc.:

```
#   ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:40:9D:2E:91:34
          inet addr:192.168.43.30  Bcast:192.168.43.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:114534 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:4810428 (285.9 KiB)
          Interrupt:6
```

For extended wireless configuration, use **iwconfig**. This tool shows wireless-only related information, like the SSID, encryption method, etc.:

```
#   iwconfig wlan0
wlan0     IEEE 802.11a/b/g  ESSID:"off/any"  Nickname:"Digi Wireless b/g"
          Mode:Managed  Frequency=2.484 GHz  Access Point: Invalid
          Bit Rate=54 Mb/s
          Retry limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Link Quality=67/100  Signal level=31/79  Noise level:0/0
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

The **Signal level** is an important parameter. The higher the number, the better the quality. For a stable connection, the signal level should be greater than 25/79. A signal level below 20/79 will result in timeouts and possibly connection losses.

The **wpa_supplicant** package also provides a tool for showing a basic status of the WLAN interface:

```
#   wpa_cli status
Selected interface 'wlan0'
bssid=00:15:c7:2b:57:a0
ssid=default
pairwise_cipher=NONE
group_cipher=NONE
key_mgmt=NONE
wpa_state=COMPLETED
ip_address=192.168.43.30
```

9.6.2. Scan the wireless network

There are several ways to scan the wireless network in search for wireless devices.

The **iwlist** command of the wireless-tools package retrieves information about the devices detected in the range of the target, such as address, protocol, bit rates, and signal quality:

```
#   iwlist wlan0 scan
    wlan0    Scan completed :
           Cell 01 - Address: 00:12:17:70:B8:8E
                     ESSID:"default"
                     Protocol:IEEE 802.11bg
                     Mode:Master
                     Channel:2
                     Encryption key:off
                     Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 6 Mb/s; 9 Mb/s
                               11 Mb/s; 12 Mb/s; 18 Mb/s; 24 Mb/s; 36 Mb/s
                               48 Mb/s; 54 Mb/s
                     Quality=67/100  Signal level=31/79
                     Extra: Last beacon: 340ms ago
```

The **wpa_cli** application scans the wireless network. This scanning is done through a two-step command, and shows basic parameters:

```
#   wpa_cli scan
Selected interface 'wlan0'
OK
#   wpa_cli scan_results
Selected interface 'wlan0'
bssid / frequency / signal level / flags / ssid
00:12:17:70:b8:8e       0        31          default
00:90:4c:60:04:00       0        17          liwi
```

9.7. Authentication and encryption

As discussed earlier, authentication is the act of confirming the identity and encryption is the process that makes information unreadable for unauthorized users. This topic lists the methods supported by the ConnectCore Wi-9C WLAN interface Linux driver:

9.7.1. Supported methods

Supported authentication and encryption methods include:

Authentication	Infrastructure mode	Ad-hoc mode
open	X	X
shared		
WPA-PSK (personal)	X	
WPA2-PSK (personal)	X	
WPA (enterprise)	X	
WPA2 (enterprise)	X	

Encryption	Infrastructure mode	Ad-hoc mode
no encryption	X	X
WEP 64/128 bits	X	
TKIP	X	
AES-CCMP	X	

9.7.2. Authentication and encryption combinations

There are several combinations of authentication and encryption methods.

Authentication	Encryption
open	no encryption
open	WEP 64/128 bits
WPA-PSK	TKIP
WPA-PSK	AES-CCMP
WPA2-PSK	AES-CCMP
WPA enterprise	AES-CCMP
WPA2 enterprise	AES-CCMP

The next topic shows how to create connections with some of these combinations.

9.8. Wireless connection examples

9.8.1. Open authentication and no encryption

9.8.1.1. Connect to an access point (infrastructure mode)

This topic explains how to connect the ConnectCore Wi-9C wireless interface to an open access point with no encryption. The different steps are done with the **wpa_supplicant** tool **wpa_cli**.

Add a network. This returns the number of this network connection:

```
#   wpa_cli add_network
Selected interface 'wlan0'
1
```

Set the ESSID of the Access Point, in this example, **default**. Replace the **1** with the number returned in the previous command, and precede and follow the double quotes with simple quotes:

```
#   wpa_cli set_network 1 ssid '"default"'
Selected interface 'wlan0'
OK
```

Set the Key management to NONE.

```
#   wpa_cli set_network 1 key_mgmt NONE
Selected interface 'wlan0'
OK
```

Select this network to work with it.

```
#   wpa_cli select_network 1
Selected interface 'wlan0'
OK
```

Alternatively, all the commands can be entered in an interactive mode, when **wpa_cli** is started without arguments. The following example does exactly the same steps done before:

```
#   wpa_cli
wpa_cli v0.4.9
Copyright (c) 2004-2005, Jouni Malinen <jkmaline@cc.hut.fi> and contributors

This program is free software. You can distribute it and/or modify it
under the terms of the GNU General Public License version 2.

Alternatively, this software may be distributed under the terms of the
BSD license. See README and COPYING for more details.


Selected interface 'wlan0'

Interactive mode

> add_network
1
> set_network 1 ssid "ndtest_wep_ap1"
OK
> set_network 1 key_mgmt NONE
OK
> select_network 1
OK
> quit
```

9.8.1.2. Connect to a computer (ad-hoc mode)

Connecting to a peer computer (ad-hoc mode) requires the same steps as shown in previous topic to connect to an Access Point (infrastructure mode), only the mode is changed.

Add a network. This returns the number of this network connection:

```
#   wpa_cli add_network
Selected interface 'wlan0'
2
```

Set the ESSID of the peer computer to connect to; in this example, **default**:

```
#   wpa_cli set_network 2 ssid '"default"'
Selected interface 'wlan0'
OK
```

Set the Key management to NONE.

```
#   wpa_cli set_network 2 key_mgmt NONE
Selected interface 'wlan0'
OK
```

Set the mode to 1 (ad-hoc):

```
#   wpa_cli set_network 2 mode 1
Selected interface 'wlan0'
OK
```

Select this network to work with it.

```
#   wpa_cli select_network 2
Selected interface 'wlan0'
OK
```

9.8.2. Open authentication and WEP encryption

To use open authentication and WEP encryption, the steps are similar to those used before, but only the WEP encryption mode is used. An open AP with WEP encryption must be used.

Add a network. This returns the number of this network connection:

```
#   wpa_cli add_network
Selected interface 'wlan0'
3
```

Set the ESSID of the Access Point, which, in this example, is **default**:

```
#   wpa_cli set_network 3 ssid '"default"'
Selected interface 'wlan0'
OK
```

Set the Key management to NONE.

```
#   wpa_cli set_network 3 key_mgmt NONE
Selected interface 'wlan0'
OK
```

Set the WEP key used in the AP. For example, if the active key is **key0** and this a 128 bit key with the values **0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD**, enter:

```
#   wpa_cli set_network 3 wep_key0 112233445566778899AABBCCDD
Selected interface 'wlan0'
OK
```

Select this network to work with it.

```
#   wpa_cli select_network 3
Selected interface 'wlan0'
OK
```

9.8.3. WPA2-PSK authentication with AES-CCMP encryption

Now use the WPA2-PSK authentication mode combined with AES-CCMP encryption. An AP with WPA2-PSK authentication and AES-CCMP encryption must be used.

Add a network. This action returns the number of this network connection:

```
#   wpa_cli add_network
Selected interface 'wlan0'
4
```

Set the ESSID of the Access Point, in this example, **default:**

```
#   wpa_cli set_network 4 ssid '"default"'
Selected interface 'wlan0'
OK
```

Set the Key management to **WPA-PSK**.

```
#   wpa_cli set_network 4 key_mgmt WPA-PSK
Selected interface 'wlan0'
OK
```

Set the protocol to **WPA2**, thus resulting in an authentication mode of **WPA2-PSK**:

```
#   wpa_cli set_network 4 proto WPA2
Selected interface 'wlan0'
OK
```

Set the pairwise ciphers for WPA to CCMP:

```
#   wpa_cli set_network 4 pairwise CCMP
Selected interface 'wlan0'
OK
```

A 256-bit PSK key must be generated basing on the AES pass phrase used in the AP, the MAC address of the AP, etc. The **wpa_passphrase** application generates this key. For example, for the pass phrase **my_pass_phrase**, execute:

```
#   wpa_passphrase default my_pass_phrase
network={
        ssid="default"
        #psk="my_pass_phrase"
        psk=8fe3077782a6262044ca53ff843bf69d143f7f158ad4d07bfd53616b4e169a5b
}
```

Set the key in the network:

```
#   wpa_cli set_network 4 psk 8fe3077782a6262044ca53ff843bf69d143f7f158ad4d07
    bfd53616b4e169a5b
Selected interface 'wlan0'
OK
```

Select this network to work with it.

```
#   wpa_cli select_network 4
Selected interface 'wlan0'
OK
```

9.9. Save the configuration

The Linux wireless configuration is done in the file **/etc/wpa_supplicant.conf**. The connection examples seen previously can be done automatically by editing this file with the correct values (see the MAN page of **wpa_supplicant.conf**).

If a configuration using the **wpa_cli** application has been created, save it to the **/etc/wpa_supplicant.conf** file with the following command:

```
#   wpa_cli save_config
Selected interface 'wlan0'
OK
```

The configuration can only be saved if the rootfs has write permissions.

9.10. Fine-tune wireless connections

The signal quality of wireless connections depends on several conditions, such as absorbability and reflections in the air, distance between target and access point, etc. The wireless driver tries to find the best bitrate, beginning with 1MB/s. If there are many more good packets than failures, the bitrate is increased. This fine-tuning process depends on the amount of data available. If only a few frames are transmitted per second, the adjusting may take a few seconds.

To establish a specific bitrate, use **iwconfig <interface> rate <bitrate>**. This example establishes a bitrate of 54 Mb/s:

```
#   iwconfig wlan0 rate 54000000
```

To list the available bitrates, use the **iwlist** command:

```
#   iwlist wlan0 rate
wlan0     11 available bit-rates :
 1 Mb/s
 2 Mb/s
 5.5 Mb/s
 6 Mb/s
 9 Mb/s
 12 Mb/s
 18 Mb/s
 24 Mb/s
 36 Mb/s
 48 Mb/s
 54 Mb/s
 Current Bit Rate=54 Mb/s
```

To return to automatic bitrate selection, enter:

```
#   iwconfig wlan0 rate auto
```

10. Boot loader development

This topic shows how to work with the boot loader. It covers how to customize, build, and install a boot loader image, then update the flash memory with the newly generated boot loader image. This topic uses the full project, **myFullProject**, created in previous topics and examples.

10.1. U-Boot projects

Working with the boot loader requires a project with U-Boot support. This is done by adding the **-u** option to the project creation script **mkproject.sh**. Since the previously created project **myFullProject** was created with that option, nothing more is required to work with the U-Boot boot loader.

The project creation script installs the whole U-Boot sources from the installed environment to the **build/U-Boot/** project subfolder. This local copy is used to build the U-Boot images.

10.2. Configure U-Boot

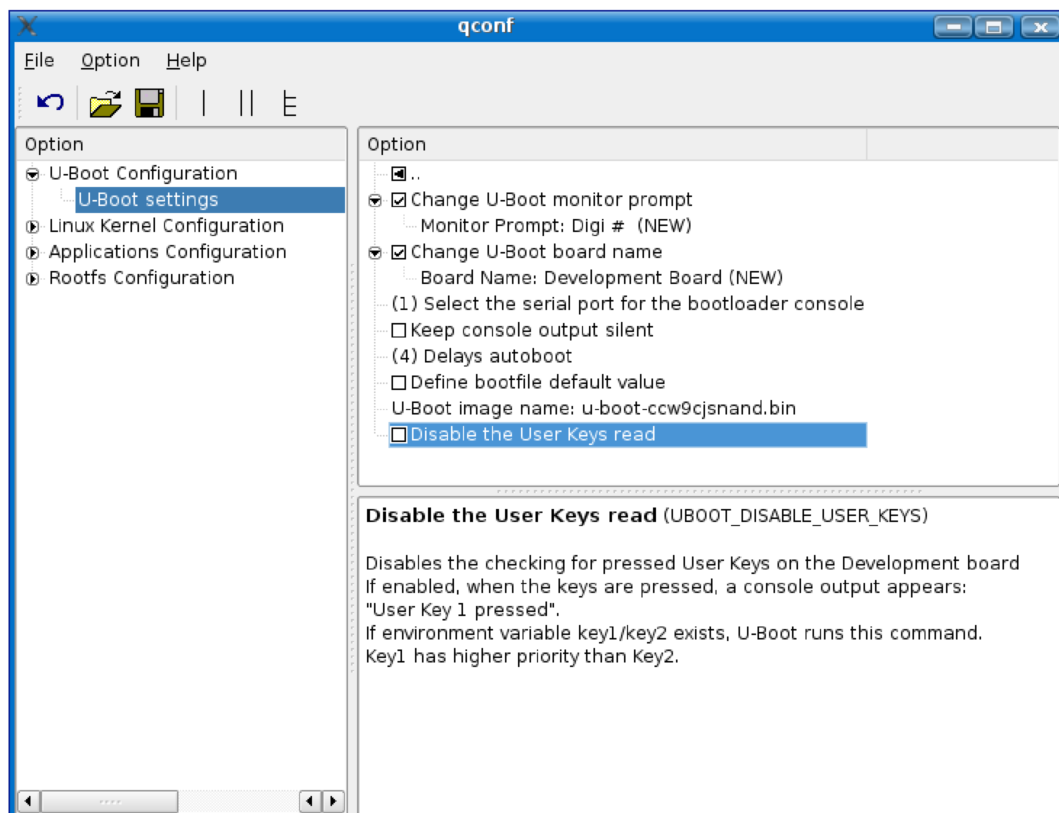
Configuring the U-Boot boot loader is done by executing the configuration tool for the project.

The project configuration tool is based on the standard U-Boot configuration tool. It is a set of Makefile rules that can be used depending on the libraries installed on the development host.

The different options for configuring U-Boot are:

```
$   make xconfig (for KDE users - qt libraries required)
$   make gconfig (for GNOME users - gtk libraries required)
$   make menuconfig (graphical configuration tool for console)
$   make config (console configuration tool)
```

For example, if **make xconfig** is used, this window is displayed:



In the U-Boot configuration tool, selecting **U-Boot Configuration > U-Boot settings** displays several U-Boot configuration settings:

- **Change U-Boot Monitor prompt**: Prompt to display in U-Boot monitor (shell).
- **Change U-Boot board name**: Board name to display in U-Boot boot message.
- **Select the serial port for boot loader console**: Configures the serial port where the U-Boot console is redirected.
- **Keep console output silent**: Disables any console output.
- **Delays autoboot**: Default number of seconds to wait before autoboot.
- **U-Boot Image name**: Name of the U-Boot image resultant of the build process.
- **Disable the User Keys read**: Disables the checking for pressed User Keys on the development board. If enabled, when the keys are pressed, a console output appears: **User Key *X* pressed**. If the environment variable ***key1/key2*** exists, U-Boot runs this command. **Key1** has higher priority than **Key2**.

10.3. Platform-specific source code

Several platform-specific source code files are involved in customizing U-Boot:

File	Description
include/configs/userconfig.h	Results from the graphical configuration tool.
include/configs/*platformname*.h	Default configuration for the module.
include/configs/digi_common*.h	Configuration for all Digi modules.
board/*platformname*/*platformname*.c	Platform initialization.

*Substitute **platformname** with the actual platform being used. To determine the platform name, see topic 1.4.*

10.4. Customize U-Boot

10.4.1. Default environment variables

U-Boot has a set of default environment variables that are defined in the environment variable **CONFIG_EXTRA_ENV_SETTINGS** in **include/configs/*platformname*.h** (see topic 1.4).

Digi has extended U-Boot by **dynamic** environment variables. They are auto-generated depending on the platform and the module U-Boot runs and are used by the commands **dboot** and **update**.

To get a list of dynamic variables and their current values, use the U-Boot command **printenv_dynamic.** The standard command **printenv** does not list them unless overwritten by user action, as seen in this example:

```
#   printenv_dynamic
rimg=rootfs-ccw9cjsnand-128.jffs2
#   printenv rimg
## Error: "rimg" not defined
#   setenv rimg rootfs-my.jffs2
#   printenv_dynamic
rimg=rootfs-my.jffs2
```

To reset dynamic variables to their default values, use the **setenv** command and set them to nothing:

```
#   setenv rimg
#   printenv_dynamic
rimg=rootfs-ccw9cjsnand-128.jffs2
```

10.4.1.1. Linux-related environment variables

Digi implements several environment variables for different OS implementations, including Linux, Windows CE, and Net+OS. Here are the specific Linux-related variables:

Variable	Description
console	Device where Linux should output the console.
ip	IP configuration to give to kernel.
kimg	Linux kernel image filename; used for kernel updates.
loadaddr	The RAM address in which to place the kernel image.
npath	Used when booting via NFS. This variable specifies the path in which the rootfs system resides on the NFS server.
rimg	Linux rootfs image filename; used for rootfs updates.
smtd	Options to give to kernel in case of a Flash or USB boot.
snfs	Options to give to kernel in case of a TFTP boot.
std_bootarg	A string of arguments passed to the Linux kernel for booting. **dboot** automatically appends the console, ip configuration, the partition table and the location where the rootfs is.
usrimg	User partition image filename; used for user partition updates.

10.5. Compile U-Boot

Once the U-Boot project element is configured, it can be built.

To build U-Boot, execute this **make** command:

```
$   make build_uboot
```

This command builds U-Boot and generates the image. The image is stored under the filename selected during configuration in the project subfolder **images/**.

*Doing a **make** without any arguments builds the complete project, including applications, kernel, kernel modules, rootfs, and U-Boot if these elements were included at the moment when the project was created.*

10.6. Install U-Boot

The final stage in using the U-Boot boot loader is installing it, which makes the U-Boot image available for the target to find.

To install U-Boot, execute this **make** command:

```
$   make install_uboot
```

This command installs U-Boot image in the **/tftpboot/** directory. There, it will be accessible for the boot loader to update via TFTP.

*Doing a **make install** installs the complete project, including every element of the project.*

10.7. Update U-Boot

10.7.1. Update from a running Linux system

If a Linux system is running on the target, the U-Boot partition can be updated with a new U-Boot image by using the **update_flash** tool, as seen in topic 7.4.2.

10.7.2. Update from U-Boot

A useful feature of U-Boot is that it can update itself. As demonstrated in topic 7.4.3, the U-Boot **update** command can directly write to the Flash memory.

Depending on the setting for the ***type*** parameter, the update command gets the image file either from a USB Flash disk or from a TFTP exposed folder in the host. The command acquires the file name from the value stored in this U-Boot environment variable:

- U-Boot image filename: **uimg**

The default value for this variable corresponds to the default image filename generated during compilation of the boot loader. If the image filename was changed, this U-Boot variable must be set accordingly.

The U-Boot **update** command handles transferring the image file to RAM, erasing the Flash sectors and writing the new image. For example, if the boot loader is in the TFTP exposed folder on the development computer, the update command is:

```
#   update uboot tftp
```

At the reboot, the new U-Boot start message will be displayed. To check it, look at the compilation date.

11. Graphics libraries

Embedded devices have limited resources and simply cannot afford the storage space or the memory to run desktop computer's graphics software.

The Embedded Linux community is working on diverse open source embedded Linux graphics system software: picoGUI, microwindows, GtkFB, Qtopia core, DirectFB, OpenGUI and many others.

As an example of what can be done, Qtopia core has been ported to the ConnectCore 9C/Wi-9C and ConnectCore 9P platforms. An LCD is required.

11.1. Qtopia core

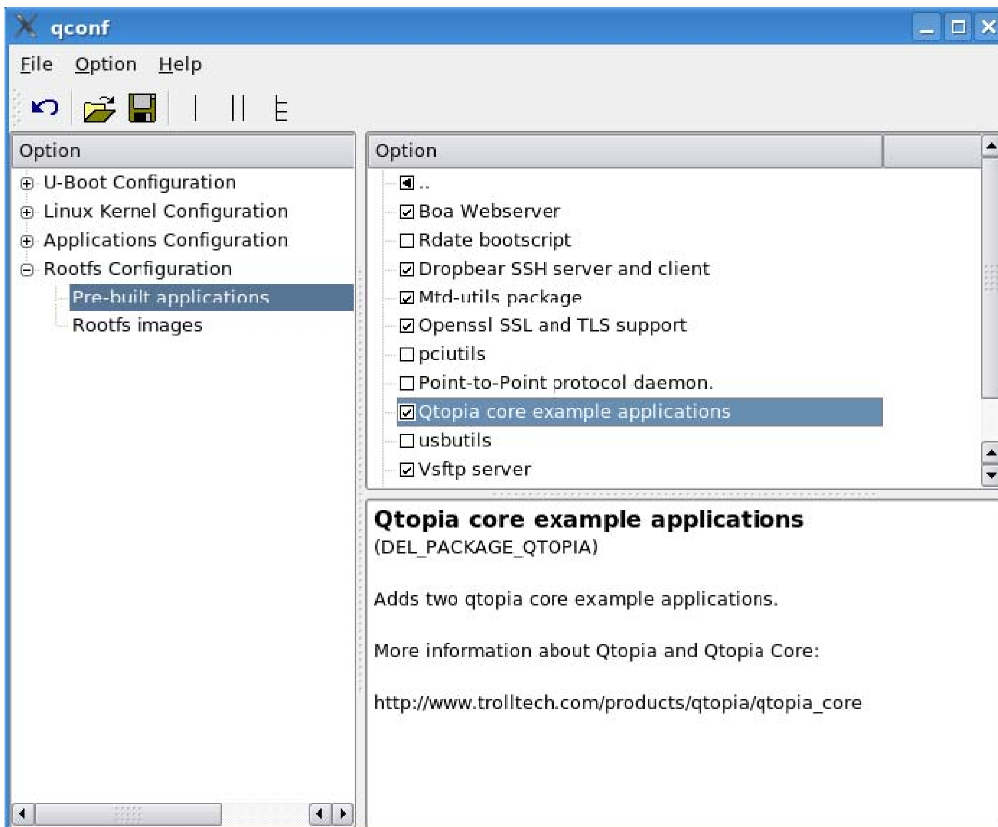
Qtopia Core, formerly Qt/Embedded, is a C++ framework for GUI and application development for embedded devices. It runs on a variety of processors, usually with Embedded Linux. Qtopia Core provides the standard Qt API for embedded devices with a lightweight windowing system.

For more information, visit the Trolltech website for Qtopia Core at http://www.trolltech.com/products/qtopia/qtopia_core

11.2. Qtopia core example applications

Projects with rootfs support can include the provided Qtopia sample applications.

Run the configuration tool from the project folder. Under **Rootfs configuration > Pre-built applications**, select the element **Qtopia core example applications**. This selection includes two applications and the Qtopia core libraries into the rootfs.



Including these applications adds around 9MB to the rootfs, specially because of the size of Qtopia libraries.

The example applications are stored into the folder **/usr/bin** of the rootfs.
To execute the **gradients** application, enter this command:

```
#   gradients
```

To execute the **spreadsheet** application, enter this command:

```
#   spreadsheet
```

12. Troubleshooting

Here are common issues and solutions when using Digi Embedded Linux.

12.1. Getting NFS service when booting with the LiveDVD

The Linux NFS server does not work with pseudo-file systems such as sysfs, tmpfs, etc. (see **http://nfs.sourceforge.net/#faq_c6**). For this reason, the NFS server cannot export any directory of the rootfs if working directly with the liveDVD.

To use the NFS server when booting directly with the liveDVD, a USB memory stick is required. This memory stick is used to export the rootfs for the target.

To mount the USB memory stick and use the NFS server from the liveDVD:

1. Boot from the liveDVD.
2. Plug in the USB memory stick.
3. Mount the USB memory stick device in **/exports** folder with:

```
$   sudo mount <device> /exports
```

4. Restart the NFS daemon with:

```
$   sudo /etc/init.d/nfs-kernel-server restart
```

5. Create projects using the **--nfs-dir=/export/nfsroot-*platformname*** option to the project creation script. Build and install the project (see topic 1.4).

Following these steps, the NFS server will work properly and the target will mount the rootfs without problems.

To unmount the USB stick

When the USB stick is no longer needed, unmount it by following these steps. It cannot be unmounted until the NFS server is down.

1. Stop the NFS server:

```
$   sudo /etc/init.d/nfs-kernel-server stop
```

2. Umount the USB stick:

```
$   sudo umount /exports
```

3. Unplug the USB stick.

12.2. Characters returned by target when target is powered off

When powering off the target, some unknown characters are echoed to the serial port.

With the power supply connected and the main power switch of the board switched off, some characters may be echoed to the serial console too, probably taking the power from the serial cable.

12.3. Writing large files to Flash from U-Boot

The **update** command in U-Boot first transfers files to RAM, then erases the Flash partition, and finally writes the files from RAM into Flash memory.

The transferred file is copied to a certain physical address in RAM, specified by the variable **loadaddr**. Therefore, the maximum length of the file to update is:

update file size limit = total RAM memory – RAM offset where the file was loaded

As a general rule, U-Boot does not allow updating a flash partition with a file whose size exceeds the available RAM memory. For example, for a module with 32MB RAM and 64MB Flash, U-Boot will not update a partition with a file that is 35MB.

Note that this limitation is due to the RAM memory size, as U-Boot first needs to transfer the file to RAM before copying it to flash.

To update files bigger than the available RAM, use the Linux **update_flash** tool (explained in topic 7.4.2).

To prevent the file from being copied to RAM before being written to Flash, do not use the **--tmpdir** option.

13. Recover a device

Normally, embedded Linux application development involves creating kernel and rootfs images. Even if bad kernel or rootfs images are written that are unable to boot, new images can be rewritten from the U-Boot monitor shell. Nonetheless, it may be best to create a custom boot loader and update it, as seen in topic 8.

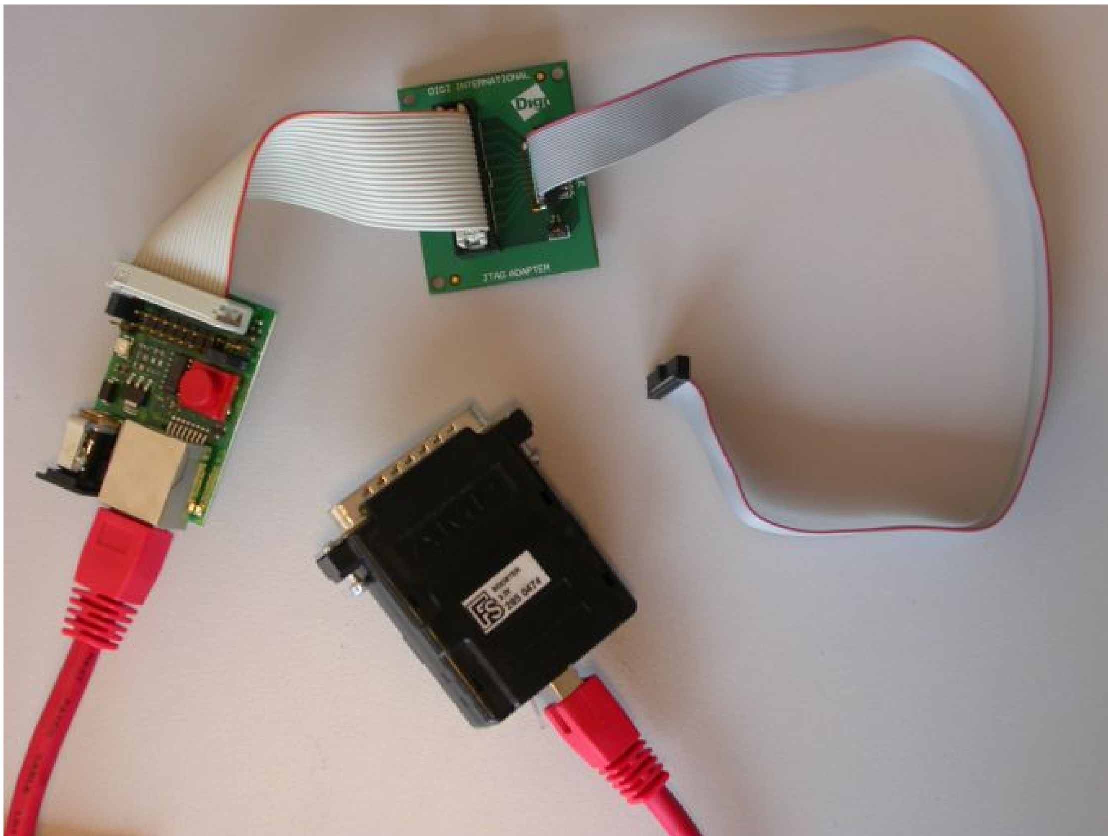
If a custom boot loader is not able to boot, or if the boot loader is erased from the Flash, a special hardware tool, called the JTAG Booster, is needed to recover this fundamental part of software.

The JTAG-Booster tool and its software are sold as a separate product. Contact your Digi distributor for purchasing information.

13.1. JTAG Booster and software

The JTAG Booster is a hardware tool with a DB25 connector in one end, to be attached to a PC, and 8 lines, known as JTAG lines, on the other. The JTAG tool comes with several cables and adaptors so that it can be plugged to the development board. The logic inside the JTAG tool allows for control over all the lines of the microprocessor via the JTAG bus.

Together with its DOS software, the JTAG Booster is used to program the Flash memory. This is useful to recover a module that is not able to boot the U-Boot boot loader anymore.



14. Uninstall Digi Embedded Linux

To uninstall Digi Embedded Linux, execute the uninstaller:

```
$   cd /usr/local/DigiEL-4.0
$   ./uninstall
```

This operation removes only the Digi Embedded Linux environment from the host. The workspace and projects remain on the disk, and must be deleted by hand.

15. References

For additional information on developing embedded Linux applications, see these resources.

Eclipse Workbench User Guide

A general guide on application development and use of the Eclipse workbench. Accessible from the Digi ESP menu; select **Help > Help contents > Additional documents > Workbench User Guide**.

C/C++ Development User Guide

A guide on development using the C/C++ Development Toolkit (CDT) plug-in of Eclipse workbench. Accessible from the Digi ESP IDE's menu **Help > Help contents > Additional documents > C/C++ Development User Guide**.

Kernel parameters

This document lists and describes Linux kernel parameters.

/usr/local/DigiEL-4.0/kernel/linux/Documentation/kernel-parameters.txt

U-Boot Reference Manual

This manual describes Digi's implementation of the U-Boot boot loader, with description of built-in commands and environment variables.

GNU make manual

This document describes the Linux **make** utility and how to write Makefiles with rules, commands, functions, etc.

http://www.gnu.org/software/make/manual/make.html

"Linux Device Drivers 3rd edition", by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

This book describes writing device drivers for the Linux system (based on kernel version 2.6.10), including char and block drivers, interrupts, DMA, memory allocation, kernel timers, etc.

ISBN: 0596005903

Digital version available at **http://lwn.net/Kernel/LDD3/**

JFFS2 file system

http://en.wikipedia.org/wiki/JFFS2

CRAMFS file system

http://en.wikipedia.org/wiki/Cramfs

ROMFS file system

http://romfs.sourceforge.net/

INITRD file system

http://en.wikipedia.org/wiki/Initrd

Serial Programming Guide for POSIX Operating Systems

http://www.easysw.com/~mike/serial/serial.html

Serial Programming HOWTO

http://tldp.org/HOWTO/Serial-Programming-HOWTO/index.html