Freescale Semiconductor
Application Note

Document Number: AN3408
Rev. 0, 02/2007

Building a Sample CGI Application

for the uClinux-Targeting ColdFire® MCF5329 Evaluation Board

by: Yaroslav Vinogradov
Roznov pod Radhostem, Czech Republic

Contents

1 Introduction

This application note describes installing and using the open-source uClinux distribution and Linux target image builder (LTIB)-based uClinux distribution for the ColdFire® MCF5329 evaluation board.

As an example, a common gateway interface (CGI) extension for the Boa web server is created. It allows sending messages to the controller area network (CAN) bus using a web browser.

1.1 What is Linux?

The operating system is the software that schedules tasks and controls the use of system resources. An operating system is made up of a kernel and various system utility programs. Linux, Microsoft Windows®, Unix®, and OS390® are examples of operating systems.

The kernel is the central part of the operating systems that manages system resources and communication between hardware and software

© Freescale Semiconductor, Inc., 2007. All rights reserved.

freescale semiconductor

components.

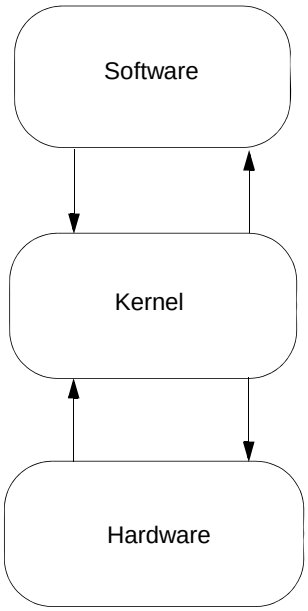

Figure 1. Communication Process

The main tasks or services provided by the kernel:

- Process management
  - Running user applications
  - Multi-tasking
  - Inter-process communication (IPC)
- Memory management
  - Allocation/de-allocation of memory for processes
  - Virtual memory addressing, translation, etc
- Device management
  - Device drivers
- System calls

Linux is a free, open-source operating system based on Unix. Linus Torvalds, with the assistance of developers from around the globe, created Linux. Linux was developed under the GNU General Public License. The source is freely available to everyone. Linux is ported to different platforms, including high-end CPUs and low-end MCUs.

1.2 What is uClinux?

uClinux is a version of Linux for “small” microcontrollers. It is a port of Linux to systems without a memory management unit (MMU). Pronounced you-see-linux, the name uClinux comes from combining the Greek letter mu (μ) and the English capital C. Mu stands for micro; C is for controller. uClinux was

first ported to the Motorola (now Freescale) MC68328 DragonBall Integrated Microprocessor. The first target system to boot successfully was the PalmPilot using a TRG SuperPilot Board with a custom boot-loader created specifically for the Linux/PalmPilot port.

The original uClinux was a derivative of the Linux 2.0 kernel intended to run on MCUs without an MMU. Originally, uClinux as an operating system was available as a large distribution in the form of source code (the distribution contains sources of the Linux kernel and many useful UserLand applications). It is widely ported to many processors for embedded systems. The uClinux kernel sources matured significantly to where it is now integrated into the main/standard Linux kernel tree. This occurred when the version went from 2.4 to 2.6. The current version of Linux/uClinux kernel is 2.6.18

uClinux has the following limitations (in comparison to other Linux-based systems):

- No memory management — No memory protection
- `fork()` system call is not implemented
- Stack does not grow automatically

If the system operates without an MMU, its stability depends on the code being well-behaved — even for user-level applications. Therefore, the developers must write very reliable code. Even a small bug in one application can make other independent applications unstable and not work.

The absence of the `fork()` system call is not significant because the `vfork()` system call is implemented. When using a `vfork()` system call, you cannot run two or more processes using the same code and data segments because the parent process is blocked until a child process calls `exec()` or `exit()`. When the `exec()` system call is executed, the process is replaced by another process (actually if the parent has called `vfork()` before, a completely new process is created after `exec()` call). The absence of the `fork()` system call is because a processor without an MMU cannot manage virtual addressing (when one physical memory address can be addressed using different virtual addresses); therefore, it cannot share a code segment transparently between different processes.

Also the stack segment on uClinux cannot grow. The stack space can be allocated only statically during the compilation of the applications. This limitation is due to the absence of the MMU unit. All processes are executed in the same address space, so when a new process is executed, there is no free hole/memory between processes. Therefore, the memory allocated for one process cannot grow; a new piece of memory can be allocated only at the start of the first available memory address. This limitation means the `brk()` system call cannot be used in applications designed for uClinux (this system call is used to grow the size of the data segment).

In uClinux, a `mmap()` system call must be used to allocate memory. Memory allocation is hidden in the C library code (`malloc()`, `free()` and other standard C functions). The uClinux implementation uses Linux system calls, so there are no problems with this limitation in most of the C code for Unix-like operating systems.

1.3 uClinux Requirements

The uClinux operating system is supported by most ColdFire MPU devices that use external memory. The minimum amount of RAM for uClinux to work is 1 MB. To allow user applications to be added, use at

least 4 MB or 8 MB. Appendix C, "uClinux Memory Map," shows the MCF5329EVB memory map. Therefore, ColdFire MCUs with limited on-chip memory are not suited for uClinux or Linux OS.

Consider the availability of drivers for the required devices. The uClinux operating system has drivers for most ColdFire devices. Some knowledge of the Linux kernel and Linux device drivers is required to develop new device drivers.

Because Linux is a large operating system and provides many services, it can be slow. For example, if real-time response is required, other pre-emptive operating systems should be considered.

The main advantage of using uClinux is its compatibility with the Linux operating system. Therefore, much of the existing application and driver code for Linux can be used with uClinux or ported to it.

2 Running a Basic uClinux System Using an Open-source Distribution

2.1 Setting up the Development System

The distribution available at http://uclinux.org/ is used as an example.

To set up the development system, the following files (packages) must be downloaded:

- Linux distribution (Tested using Red Hat Enterprise Workstation 3.0)
- GNU C/C++ compiler — http://uclinux.org/pub/uClinux/m68k-elf-tools/tools-20060615/m68k-uclinux-tools-20060615.sh
- uClinux distribution — http://uclinux.org/ports/coldfire/uClinux-dist-20051110.tar.bz2
- One of the latest Linux kernels (version 2.6.17.1) — http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.17.1.tar.bz2
- uClinux kernel patch — http://uclinux.org/pub/uClinux/uClinux-2.6.x/linux-2.6.17-uc0.patch.gz
- Set of uClinux kernel patches to be used with MCF5329
- Terminal program (Minicom can be used with Linux, Tera Term Pro with Windows)

NOTE

Linux must be installed on your PC. This application note does not explain installation.

The next step is installing the GNU C/C++ compiler. The precompiled binary version of GCC version 4.1.1 is available on the uClinux website. To install it, you need root privileges. To install, run the following command at the command prompt from the directory where you placed the downloaded files:

```
$ su –c './m68k-uclinux-tools-20060615.sh'
```

After a few minutes, the binaries of the GCC compiler for ColdFire are installed into `/usr/local/bin` and `/usr/local/m68k-uclinux` directories. The prefixes for the compilers are `m68k-elf-` and `m68k-uclinux-` . For ColdFire, the command is `m68k-elf-gcc` to compile a stand-alone application or uClinux kernel, and `m68k-uclinux-gcc` to compile uClinux applications.

Table 1. Compilers for Application Types

Application (app)	Compiler Prefix	Path to compiler
Stand alone app or kernel	`m68k-elf`	`/usr/local/bin`
uClinux app	`m68k-uclinux`	`/usr/local/m68k-uclinux`

Also, the path to the newly installed GCC compiler must be added to the `$PATH` environment variable. If the bash shell is used, you can do this by editing `~/.bash_profile` file. Add the following lines to it:

```
PATH=$PATH:/usr/local/bin
export PATH
```

The line that sets the `$PATH` environment variable could be present. If so, the line should be edited to include `/usr/`local/bin. Also, the export line with `$PATH` variable may already be present. In this case, it is not required to edit the export line.

To install the uClinux distribution, unpack the downloaded `tar.bz2` archive. For this, two standard Unix utilities are used: `tar and bzip2`. The single command for uncompressing and unpacking is:

```
$ tar jxvf uClinux-dist-20051110.tar.bz2
```

This command must be executed in the directory where the `uClinux-dist-20051110.tar.bz2` file is, and it unpacks the archive into the current directory. After unpacking, in the current directory the uClinux-dist directory is created holding the archive contents.

This uClinux distribution has Linux kernels version 2.4.x and 2.6.x. However, the 2.6.x kernel included in the distribution is quite old. Download and use a newer version.

If you want to use a newer Linux kernel version (like 2.6.17.1), unpack the downloaded kernel using the following command:

```
$ tar jxvf linux-2.6.17.1.tar.bz2
```

This command unpacks the Linux kernel into the current directory. All the kernel sources are placed in the linux-2.6.17.1 directory.

```
$ rm –rf linux-2.6.x
```

Then, insert the new Linux kernel into the uClinux distribution. This can be done by copying the content of the `linux-2.6.17.1` directory into the `linux-2.6.x` directory in the uClinux-dist (this directory must be created because it was removed during the previous step). You can also insert the Linux kernel by creating the symbolic link named `linux-2.6.x` to the `linux-2.6.17.1` directory by running the following command in the `uClinux-dist` directory. It is assumed that `linux-2.6.17.1.tar.bz2` was unpacked in the same top level directory as `uClinux-dist-20051110.tar.bz2`:

```
$ ln –s ../linux-2.6.17.1 linux-2.6.x
```

Because the Linux kernel version 2.6.17.1 does not have all the required changes in source code (kernel and drivers), a set of uClinux-and MCF5329-specific patches must be applied to run on the MCF5329. To apply one patch file, the following command can be executed in the `uClinux-dist/linux-2.6.x` directory:

```
$ patch –p1 <patch_file_name
```

Instead of `patch_file_name`, the full path to the unpacked (using `gunzip patch_file`) patch file must be given. You can patch the kernel without unpacking the patch file by running the following command:

```
zcat patch_file_name | patch –p1
```

Instead of `patch_file_name`, in this case, the full path to the packed patch file must be supplied. For example, to apply the uClinux patch `linux-2.6.17-uc0.patch.gz`. You can use the following command (here it is assumed that the downloaded patch is placed in the same directory as other `tar.bz2` archives). First, the `linux-2.6.17-uc0` patch must be applied, then all the others in any order.

The following command can be used to apply the patch file. To complete the process, use it with all other patch files (change the command to invoke the relevant patch file name) to complete the process.

```
zcat ../../linux-2.6.17-uc0.patch.gz | patch –p1
```

2.2 Configuration of uClinux Distribution

Before uploading the image of the Linux kernel and UserLand applications, the image must be configured and built. The uClinux distribution has a set of configuration scripts that allow configuring the uClinux kernel and UserLand applications. Configuration is possible using text-based menus and graphical menus. To start a text-based configuration, execute `$ make menuconfig` in uClinux-dist directory:

The displayed menu looks like the next figure:

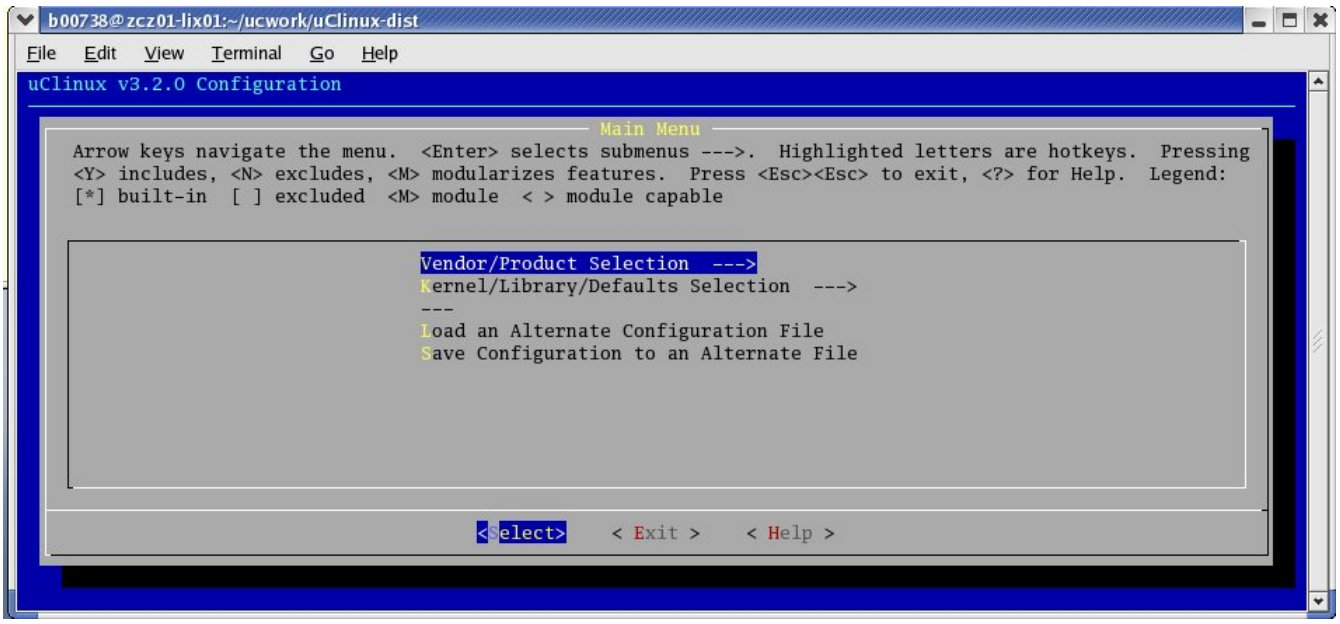

Figure 2. uClinux Distribution Main Configuration Menu

To run a graphical configuration, execute `$ make make config`.

The first menu of the uClinux configuration allows you to choose the global defaults:

- `Vendor/Product selection`
- `Kernel/Library/Defaults Selection`

Select `Freescale` as the vendor and `M5329EVB` as the product.

Submenu `Kernel/Library/Defaults Selection` gives more choices.

- At `Kernel Version`, select `linux-2.6.x` because the 2.6.x version of Linux kernel is used.
- At Libc Version select `uClibc`. This is the most stable and latest implementation of LibC for uClinux.
- Select the `Default all settings (lose changes)` checkbox. This should be selected only the first time the configuration process is executed to take the vendor default settings for the kernel and the set of UserLand packages.
- Select the `Customize Kernel Settings` check box. This leads to a display menu for customizing kernel settings. Select `Customize Vendor/User Settings`. This allows you to choose the set of land programs to be included in the image.
- To update the default settings of the Linux kernel and the set of UserLand programs for the current vendor, choose `Update Default Vendor Settings`. In this case, all the current settings are saved as defaults for the current vendor.

2.3 uClinux Kernel Configuration

The next menu displayed is the Linux Kernel Configuration:

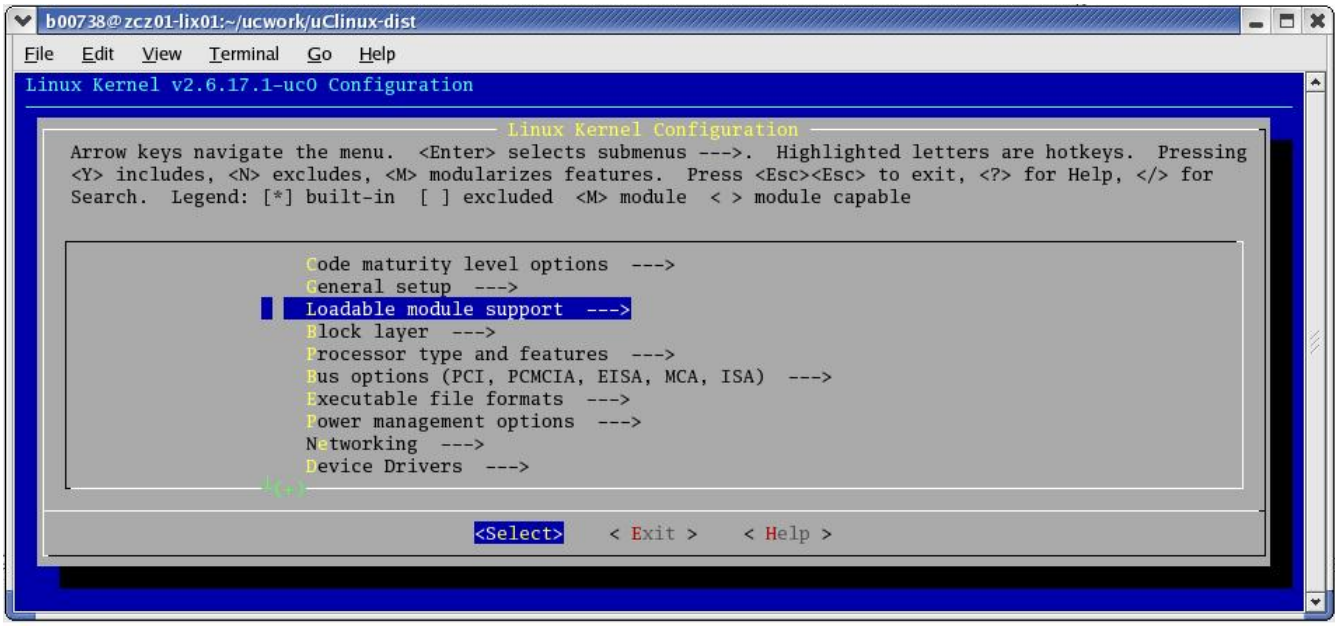

Figure 3. Linux Kernel Configuration Menu

By configuring the kernel, you can select an exact set of required features and drivers. There are many menus and sub-menus to go through during kernel configuration. For most selections, help messages appear. Let us examine the required set of options regarding the MCF5329.

The first submenu is `Code maturity level options`. It has only one item: `Prompt for development and/or incomplete code/drivers`. This option must be selected. It allows us to display options for experimental features.

The next submenu is `General setup`. It contains options for configuring general kernel features. For the MCF5329, select the following options:

- `System V IPC` — Enables support for inter-process communications
- `Enable 16-bit UID system calls` — Enables a set of legacy 16-bit UID syscalls wrappers. This option is required because the uClibC library uses it.

Then `Configure standard kernel features (for small systems)` must be selected because we are building a Linux kernel for an embedded system. This then allows access to kernel options for embedded systems. The following options must be selected:

- `Enabled support for printk` — Allows printing debug information from drivers, etc.
- `BUG() support` — Reporting information about fatal conditions/errors. This option should be disabled only if the embedded system does not have any way to print messages anywhere; in this way, you can slightly reduce the size of the compiled kernel code.
- `Enable ELF core dumps` — Enables core dumps in case of errors.
- `Enable full-sized data structures for code` — If disabled, the size of miscellaneous core kernel data structures is reduced. This may reduce performance.
- `Use full SLAB allocator` — Safer memory allocator

The next submenu is `Loadable module support`. Many Linux kernel drivers can be compiled as loadable modules. This allows loading only the required drivers at runtime and, therefore, reduces the size of the compiled kernel code. You must select `Enable loadable module support`. Also, select `Module unloading` if you plan to unload modules while running the kernel.

The next main submenu is `Processor type and features`. Options allow choosing the CPU and memory map used on the target board. For the MCF5329, the `CPU` should be `MCF532x`.

You must select `Enable setting the CPU clock frequency`. You must set `Set the core clock frequency` to `240000000` because the MCF5329 CPU on the MCF5329EVB operates at 240 MHz.

You must set `Set the core/bus clock divide ration` to `3` because the device bus clock on the MCF5329 is core clock divided by 3 (80 MHz). You must select `Freescale M5329EVB board support` because the kernel for this EVB is being built. The following values should be set as `RAM configuration`:

- `Address of the base of RAM` = `0x40020000`
- `RAM size (in bytes)` = `0x1FE0000`
- `Address of the base of system vectors` = `0x40000000`
- `Address of the base of kernel code` = `0x40020000`

Set `RAM bus width` to `AUTO`. Set `Kernel executes from` to `RAM`. `Memory model` should be `Flat memory`

The next submenu is `Bus options (PCI, PCMCIA, EISA, MCA, ISA)`. Because the MCF5329 processor does not have any such buses, do not select anything there.

The next submenu is `Executable file formats`. Flat binary format is the most common executable file format uClinux operates with. Therefore, `Kernel support for flat binaries` should be selected.

The next main submenu is `Networking`. It allows configuration of network protocols (such as TCP/IP) and their options. `Networking support` must be selected, and in the `Networking options` submenu, select `Packet socket`, `Unix domain sockets`, `TCP/IP networking`.

One of the largest submenus with further submenus is `Device drivers`. All available device drivers are selected here. Most of the drivers can be built as a module, part of the kernel, or not at all. For this example, select all the suggested options to be built as the part of the kernel (not as module) to simplify loading of them.

`Memory Technology Devices (MTD)` represents the subsystem of drivers for devices accessed as memory such as flash chips, or even part of regular RAM memory used to store file system files. Because the image of the file system with UserLand programs is taken from the memory drive (in RAM), this option should be enabled. The following options should also be enabled:

- `MTD partitioning support`
- `Direct char device access to MTD devices`
- `Caching block device access to MTD devices`

Inside the `RAM/ROM/Flash chip drivers` submenu, enable `Support for RAM chips in bus mapping`. In the `Mapping drivers for chip access` submenu, enable `Generic uClinux RAM/ROM filesystem support`.

The MCF5329EVB board has NAND flash chips on it. To enable a driver to support them, the following options must be enabled (in the `NAND Flash Device Drivers` submenu):[1]

- `NAND device support` — To have generic support for NAND flash devices
- `NAND flash device on M5329 board` — To have the board-specific driver

Inside the `Block devices` submenu, select the following options:

- `Loopback device support`
- `Network block device support`
- `RAM dist support` — Allows creating a writable file system in RAM. `Default Number of RAM disks` should be 16, and `Default RAM disk size (kbytes)` should be `8192`.

`SCSI device support` options for the case described in this application note are not required. But `SCSI disk support` option is required to mount USB flash drives (for example).

In the `Network device support` submenu, enable the following options; because the FEC is present on the MCF5329, the support for it must be enabled to use it:

- `Network device support` — To have basic network devices support code
- Inside `Ethernet (10 or 100 Mbit)`, enable `FEC ethernet controller (of ColdFire CPUs)`

Look at the `Character devices` submenu. Enable the following options:

- `Virtual terminal`
- Inside `Serial drivers`, enable `ColdFire serial support`
- `Legacy (BSD) PTY support`

1. The silicon mask sets prior to 3M29B have documented errata for programmable signal timing used for NAND flash interface.

- `ColdFire FlexCAN module support` and `CAN0` — To have CAN4Linux drivers for the FlexCAN module on MCF5329

This example does not use the I^2C bus, but if it is needed, support for it can be enabled under `I2C support`. The same is true for the SPI bus (it can be enabled under `SPI support`).

If the support of the graphics frame buffer driver for ColdFire is required, it should be enabled in the `Graphic support` submenu. To do this, you must enable the following options (not required for this example):

- `Support for frame buffer devices`
- `ColdFire MCF532x Framebuffer`
- The resolution can be selected (`240x320, 640x480, 800x600`)
- Under `Logo configuration` display of the Linux penguin logo can be configured

Similarly, different USB-related options can be chosen in the `USB support` submenu.

The next large submenu is `File systems`. This submenu allows configuration of the available file system drives. Almost any file system can be used with almost any correct block devices (block of memory in RAM, flash devices, USB storage device, etc.). Select the following file systems:

- `Second extended fs support`
- `ROM file system support`
- `DOS/FAT/NT File systems` — If you want to use a USB stick written under Windows
- Under `Pseudo filesystems` select: `/proc file system support` and `sysfs file system support`. Sysfs is a virtual file system provided by 2.6.x Linux kernels. Sysfs exports information about devices and drivers from kernel device model to userspace, and is also used for configuration.[1]
- Under `Miscellaneous filesystems` JFFS2 file system (`Journaling Flash File System v2 (JFFS2) support`) should be selected. This is a very efficient file system designed for flash memories with compression support. It should be selected and loaded as a module if the ROMFS file system or NFS file system is used as the root-file system
- Under `Network File Systems` support for network file systems can be enabled. It allows the board to be a server or client using NFS protocols, or to mount Windows using the SMB protocol
- Under `Native Language Support` the following options must be enabled if a FAT file system is used (for example for USB sticks). It is: `Codepage 437` and `NLS ISO 8859-1`

That completes the main options for configuring a Linux kernel on the MCF5329.

2.4 UserLand Application Configuration

The next step is configuring the set of UserLand applications. The Linux kernel itself is not too useful (as with any kernel). It simply provides services to be used by applications such as shells, web servers, text editors, etc. The uClinux distribution comes with a rich set of UserLand applications. Also, new specific applications can be developed. We look at a minimum set of applications.

The menu for configuration of UserLand applications is displayed after kernel configuration menu if `Customize Vendor/User Settings` option is selected at the very first step. You can skip the kernel

1.http://en.wikipedia.org/wiki/Sysfs

configuration menu if the `Customize Vendor/User Settings` option is selected and the `Customize Kernel Settings` option is not selected.

The following menu is used to configure the UserLand applications:

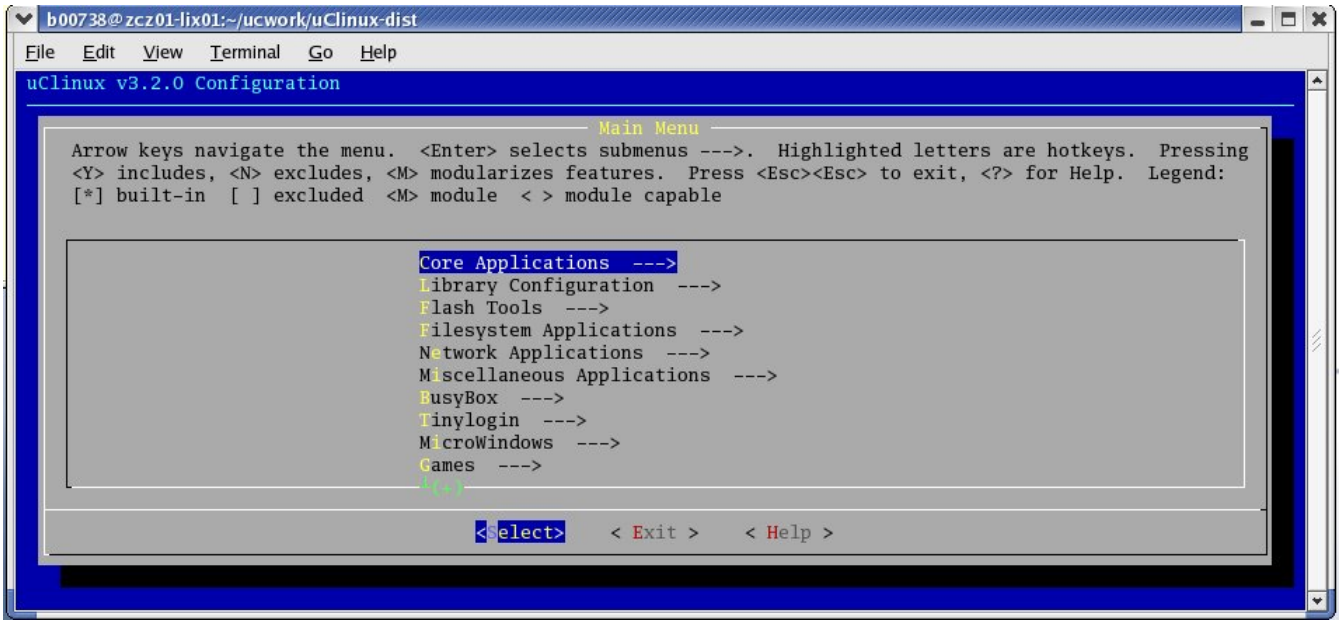

Figure 4. uClinux Application's Configuration Menu

The UserLand applications are divided into many groups.

The first group is `Core Applications`. It represents the applications that must run, such as the first process executed by the Linux kernel called `init` which manages all other processes. Select the following options from this group:

- `init` — Init process
- `enable console shell` — Asks init to start a console shell after processing all configuration/startup scripts
- For a shell, select the simplest and smallest shell called `Sash`. There are also many other shells supported such as `minix, nwsh, bash, msh`, etc.
- `expand` — Simple program to unpack archives

There are many basic Unix programs that can be selected such as `cron`, `at`, `passwd`.

The next group is called `Library Configuration`, which allows a choice of different libraries such as flex, secure socket layer (SSL), etc. No special libraries are used in this example.

The next group is called `Flash Tools`, which contains programs to write/erase different flash drivers. This example does not use them, but for example, `mtd-utils` could be selected to have tools available to erase/write the NAND flash available on the board.

The next group is called `Filesystem applications` and contains tools to access different file systems. They allow the creation and reading/writing of different file systems. This group has tools such as `mount` to

mount flash or USB drives or NFS file systems, or tools to mount Windows network sharing using the SAMBA protocol.

The next group called `Network applications` provides network-related applications such as web servers, ftp servers, and clients, etc. For our application, select:

- `boa` — Simple web server for embedded systems
- `ping` — Test program for networking (not needed by our example but very useful to test if the network is configured and working properly)
- `ifconfig` — To configure the Ethernet interface in the kernel (that is, to assign network address, network mask, etc.)
- `route` — To configure the network routing table inside the kernel

The next large group is called `Miscellaneous Applications`. All other applications like the `vi` text editor are present. For our example, no applications from this group are used.

The next group is `Busybox`. This tool is an all-in-one implementation of Unix command-line tools like `ls`, `cp`, etc. All the tools are in one binary image; therefore, the required size in the root-file system is greatly reduced. This tool is used mainly for the embedded systems and rescue disks.

The next group is `TinyLogin`. It is an implementation of the login mechanism for embedded systems. Before getting to the shell, the user name and password for user authentication is asked if this tool is used. Our application does not use it.

The next group is `Microwindows`. It is a set of simple graphical applications (such as terminal, etc.) and a useful library for developing graphical applications for embedded systems. This application/library can be used on the MCF5329 to display information in a graphical manner using the frame buffer Linux kernel driver. Our application does not use it.

The next group is `Miscellaneous Configuration`. It allows, for example, you to add the startup script to the target file system which creates a RAM drive to store temporary files during work. Please select `512K RAMFS image` and `generic cgi`. `generic cgi` is an application to test the Boa web server is working.

The next step is to build everything that was selected. It can be done very simply. Just run the following command in `uClinux-dist` directory:

```
$ make
```

After this, the first Linux kernel is built. Then, if there were no errors, the uClibc and then the UserLand applications are built. If everything goes without errors, the image that contains the Linux kernel and ROMFS root-file system are built. This image is called `image.bin`. By default, the image, as a final step, is copied to the `/tftpboot` directory to download it to the board using a TFTP server. Before the build, you must have enough rights to create `image.bin` file in the `/tftpboot` directory. Appendix D, “Setting up the TFTP Server,” describes the steps required to configure the TFTP server and the terminal application.

2.5 Configuring and Using dBUG

On the board you should have dBUG v4a.1a.1c or later. Version 4b.1a.1d has extra features.

The following steps should be done to configure the dBUG:

1. At the dBUG command prompt run `show`
2. Customize the dBUG for your system:

```
set baud 115200
set server <Linux Host IP address>
set client <board IP address>
set gateway <gateway IP address>
set netmask <net mask>
set filename image.bin
set filetype image
set kcl rootfstype=romfs – this is needed in case you have new dBUG
```

3. Verify the settings using the `show` command

To download and run the uClinux image, the following dBUG commands are used (the memory map for the dBUG is shown for the MCF5329EVB in Appendix B, “dBUG Memory Map” — from it we can get the 0x40020000 address):

```
dBUG> dn
dBUG> go 0x40020000
```

The first command is used to download the image using TFTP protocol. The second command executes the downloaded image at address `0x40020000`.

If everything works, the following messages are displayed on the terminal:

```
Linux version 2.6.17.7-uc1 (b00738@zcz01-lix01) (gcc version 4.1.1) #19 Mon Nov
6 14:55:34 CET 2006
uClinux/COLDFIRE(m532x)
COLDFIRE port done by Greg Ungerer, gerg@snapgear.com
Flat model support (C) 1998,1999 Kenneth Albanowski, D. Jeff Dionne
Built 1 zonelists
Kernel command line: rootfstype=romfs
PID hash table entries: 128 (order: 7, 512 bytes)
Console: colour dummy device 80x25
Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
Memory available: 26656k/32640k RAM, (1252k kernel code, 264k data)
Mount-cache hash table entries: 512
NET: Registered protocol family 16
USB-MCF532x: (OTG module) UDC device is registered
SCSI subsystem initialized
NET: Registered protocol family 2
IP route cache hash table entries: 256 (order: -2, 1024 bytes)
TCP established hash table entries: 1024 (order: 0, 4096 bytes)
TCP bind hash table entries: 512 (order: -1, 2048 bytes)
TCP: Hash tables configured (established 1024 bind 512)
TCP reno registered
io scheduler noop registered
io scheduler cfq registered (default)
Initing M532x Framebuffer
Console: switching to colour frame buffer device 80x30
fb0: M532x FB frame buffer device
ISA-Philips-Basic-CAN memory mapped CAN Driver 3.0_ColdFire_FlexCAN (c) Nov  3 2
006 11:02:21
```

```
ColdFire internal UART serial driver version 1.00
ttyS0 at 0xfc060000 (irq = 90) is a builtin ColdFire UART
ttyS1 at 0xfc064000 (irq = 91) is a builtin ColdFire UART
RAMDISK driver initialized: 16 RAM disks of 8192K size 1024 blocksize
loop: loaded (max 8 devices)
nbd: registered device at major 43
FEC ENET Version 0.2
fec: PHY @ 0x1, ID 0x20005c90 -- DP83848
eth0: ethernet 00:cf:53:29:cf:01
uclinux[mtd]: RAM probe address=0x4019b300 size=0x412000
Creating 1 MTD partitions on "RAM":
0x00000000-0x00412000 : "ROMfs"
uclinux[mtd]: set ROMfs to be root filesystem
QSPI: spi->max_speed_hz 300000
QSPI: Baud set to 133
COLDFIRE-QSPI: probed and master registered
i2c /dev entries driver
TCP bic registered
NET: Registered protocol family 1
NET: Registered protocol family 17
VFS: Mounted root (romfs filesystem) readonly.
Freeing unused kernel memory: 56k freed (0x4016b000 - 0x40178000)
Shell invoked to run file: /etc/rc
Command: hostname uClinux
Command: /bin/expand /etc/ramfs.img /dev/ram1
Command: mount -t proc proc /proc
Command: mount -t ext2 /dev/ram1 /var
Command: mount -t sysfs sysfs /sys
Command: mkdir /var/tmp
Command: mkdir /var/log
Command: mkdir /var/run
Command: mkdir /var/lock
Command: mkdir /var/empty
Command:
Command:
Command:
Command:
Command: ifconfig lo 127.0.0.1
Command: route add -net 127.0.0.0 netmask 255.0.0.0 lo
Command: dhcpcd -p -a eth0 &
[17]
Command: cat /etc/motd
Welcome to
          ____ _  _
         /  __| ||_|
    _   _| |  | | _ ____  _   _  _  _
   | | | | |  | || |  _ \| | | |\ \/ /
   | |_| | |__| || | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting

Sash command shell (version 1.1.1)
/>
```

3 Running a Basic uClinux System Using Linux-target-image-builder (LTIB)-based Distribution

3.1 Development System Using LTIB

The LTIB is a Freescale open-source tool. It is written using Perl scripting language and simplifies installation, compilation of Linux kernels, and other UserLand programs for Freescale MPUs. In other words, LTIB can help develop and deploy board-support packages (BSPs) for various target platforms. By using this tool, you can develop a GNU/Linux image for the target platform.

The BSP for the MCF5329 is available on the Freescale web site in the form of an ISO image. This image can be written to a CD or used as a file within Linux. To install the LTIB BSP for MCF5329, follow these steps:

1. As root user, mount the ISO image (if you use a file) or CD-ROM on your machine:
   ```
   # mount –o loop m5329_evb_20060705-ltib.iso <mount-point>
   ```
2. As non-root user, install the LTIB:
   ```
   <mount-point>/install.sh
   ```

NOTE

You are prompted to input the desired LTIB install path. You must have the correct permissions for the install path.

There are no uninstall scripts. To uninstall LTIB just remove the `/opt/freescale/pkgs`, `/opt/freescale/ltib`, and `<install-path>/ltib` directories.

After installation, run:

```
$ cd <install-path>/ltib
$ ./ltib
```

During LTIB installation and after the first run, the kernel sources, sources of UserLand packages, and GCC compiler are installed. Also, pre-built binaries of the Linux kernel and default UserLand packages are installed, but they can be rebuilt using LTIB later, if required.

3.2 Using LTIB

The following command can be used to configure kernel and UserLand packages using LTIB:

```
$ ./ltib –c
```

This command displays the LTIB configuration menu shown on the next figure. You can select the C library type, toolchain, kernel version, package list for root-file system, and user-configurable options for the root-file system.

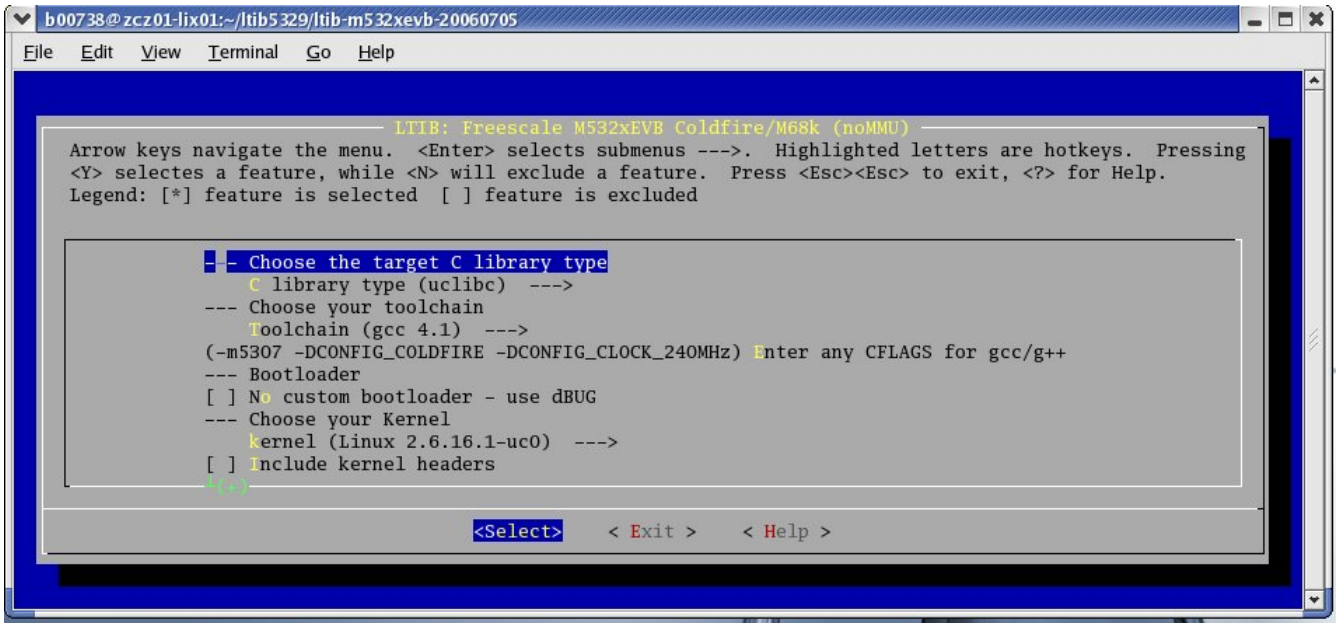

Figure 5. LTIB Configuration Menu

As a C library type, only the uClibc library is supported; it is the only choice.

For the toolchain, you can use the bundled LTIB for MCF5329 GNU GCC version 4.1 or custom toolchain. There are two options under `Toolchain` menu: `gcc 4.1` and `custom`. Choose `gcc 4.1` for this example.

You can also pass command-line arguments to GCC. This is possible through the `Enter any CFLAGS for gcc/g++` entry. As default, these are: `-m5307 -DCONFIG_COLDFIRE -DCONFIG_CLOCK_240MHz`. Leave it this way for this example.

There are no custom bootloaders supported for MCF5329. Do not select `No custom bootloader – use dBUG`.

For the Linux kernel there are three possible selections:

- `Linux 2.6.16.1-uc0` — Default uClinux kernel supplied with this LTIB package
- `Local Linux directory built` — Allows you to use the kernel from a local source directory (actually any version for MCF5329)
- `Don't build the Linux kernel` — Kernel is not built

Select `Linux 2.6.16.1-uc0` for this example.

`Include kernel headers` allows you to include the kernel headers to the root-file system. This is useful, for example, to run GCC directly on the target but is not needed for our example.

You can also specify the predefined configuration for the Linux kernel. The following predefined configurations are available:

- `linux-2.6.x.config` — Default configuration. Almost all available drivers are enabled, and the NFS root-file system is supported as a root-file system. Also, if options are changed, they are saved in this configuration for further use.
- `nfs-2.6.x.config` — Same as default configuration. Can also be used to return to the default values.
- `romfs-2.6.x.config` — Same as default configuration, but NFS support is disabled. The ROMFS root-file system image is used instead. This image is appended to the kernel image during building.

`Configure the kernel` is the next option and allows fine-tuning the kernel options before compilation.

The next main submenu is called `Package list`. All UserLand packages are displayed. For our example, select the following packages:

- `boa` — Simple web server
- `busybox` — Shell and other small Unix tools replacement
- `can4linux` — can4linux test applications
- `skeleton base files` — Skeleton root-file system

The next submenu is `Options` after `Target System Configuration`. These allow configuring a root-file system to have user-specific values such as network address, network mask, etc. These options must be changed to use an NFS root-file system.

There are the following options:

- `target hostname` — Name of the board's host (Linux PC)
- `boot up with a tty and login` — If selected, a login name and password is requested when the Linux on the board starts. Leave this option unchecked for this example.
- `start networking` — Start the networking scripts. Check this option.
- Under `network setup` you can set up network parameters. Check the `Enable interface 0` as `interface` type `eth0`. Then, you can get all settings using the DHCP protocol (for this, check out `get network parameters using dhcp`) or specify them manually. It is better to specify them manually. For this, you need the following parameters of your board: IP address, netmask, broadcast address, gateway address, and name-server IP address.
- `set the system time at startup` — If checked, the board tries to connect to an available Internet time server to set the current time on the board. Not required for this example.
- `start syslogd/klogd` — Start the syslogd and klogd daemons. Not required for this example.
- `start inetd` — Starts inetd daemon. Not required for this example.
- `start boa (webserver)` — Starts Boa web server. Check this option for this example.
- command line arguments for Boa — Specify here only `-c /etc` for our example.

The next options are under `Options` after the `Target Image Generation` menu. You can now choose the root-file-system type:

- `NFS only` — Root-file system is mounted over NFS from the Linux host.
- `romfs (uClinux)` — Root-file system is built as a ROMFS file system and then appended to the Linux kernel image.

- `jffs2` — Root-file system built as `jffs2` image.
- `ext2.gz ramdisk` — Root-file system built as an ext2 file system and packed by gzip to unpack it to a RAM disk.

For our example, the `NFS only` option is used.

When the `Exit` option is chosen, LTIB asks if the new settings should be saved. After saving the configuration, everything is built using the selected options. If everything proceeds correctly, at the end a `Build Succeeded` message is displayed.

The TFTP server must be running on the Linux host to download the Linux kernel to the board. Refer to Appendix D, “Setting up the TFTP Server”.

The next step is to configure the NFS server and export an NFS root-file system built by LTIB. Refer to Appendix E, “Setting up the NFS Server”.

3.3 Configuring and Using dBUG for NFS Root-file System

On the board, you should have dBUG v4a.1a.1c or later. Version 4b.1a.1d has extra features required to use NFS root-file system.

- At the dBUG command prompt, run show.
- Customize the dBUG for your system:

```
set baud 115200
set server <Linux Host IP address>
set client <board IP address>
set gateway <gateway IP address>
set netmask <net mask>
set filename image.bin
set filetype image
set kcl root=/dev/nfs rw nfsroot=<Host IP>:/tftpboot/ltib/rootfs ip=<Target IP>:<Host
IP>:<Gateway IP>:<Netmask>::eth0:off
```

- Verify the settings using show command.

For the NFS, carefully set up the kcl parameter. Change everything marked in <> to the values specific for your configuration.

To download and run the uClinux image, the following dBUG commands are used:

```
dBUG> dn
dBUG> go 0x40020000
```

The first command downloads the uClinux kernel using TFTP protocol. The second runs it. During startup, the kernel mounts the NFS root-file system.

4 Using Boa Web Server

Boa is a small single-tasking HTTP server. Unlike other traditional web servers, it does not create new processes or threads to handle multiple connections. It internally multiplexes all HTTP connections and

forks (vfork) to run only CGI programs (which must be separate processes). The primary design goals of the Boa are speed and security. More information about the Boa web server is available from the http://www.boa.org website.

4.1 Running the Boa Web Server when Using uClinux Distribution

The uClinux distribution comes with the Boa web server. The Boa web server is executed using init process and is defined in the /etc/inittab file. You can also run it from the command line. The simple Boa server configuration is placed in the /home/httpd folder on the target root-file system in a file called boa.conf. Any HTML files can be placed into the /home/httpd folder, and CGI extensions into /home/httpd/cgi-bin.

The networking on the board must be configured to access the Boa web server from a web browser on another host. This can be done in two ways:

- Using DHCP protocol. For this, a DHCP client application must be enabled during the uClinux configuration and then executed. The command for execution is:

  ```
  $ dhcpd –p –a eth0 &
  ```

  This command can be executed from a startup script placed in the uClinux-dist/vendors/Generic/big/rc file or from the board console
- To manually specify the IP address of the board and other network parameters, the ifconfig and route command line tools are required and must be selected during uClinux configuration. The appropriate commands are:

  ```
  $ ifconfig eth0 <ip-addres> netmask <netmask> gw up
  $ route add default gateway <gateway-ip> eth0
  ```

  The first command brings up the Ethernet interface eth0 with the specified parameters. The second command adds a default routing rule to the default gateway host. The second command is not required if the board and host that accesses the board are on the same network. These commands can be executed manually each time the uClinux on the board starts, or they can be appended to the uClinux-dist/vendors/Generic/big/rc file.

When the network interface is up and running, it can be tested using the ping command:

```
$ ping <Linux-Host-IP>
```

Now, you can access the test page on the Boa web server. Type the http://<board-ip> URL into the web browser. The basic test web page should be displayed.

4.2 Running Boa Web Server When Using LTIB

LTIB also comes with the Boa web server. However, the version available in LTIB 20060705 is not stable when working with CGI extensions. Therefore, a newer version should be used. Place the boa-0.94.14rc21.tar.gz file and the boa-0.94.14rc21-nommu.patch files into the /opt/freescale/pkgs folder. The boa-nommu.spec file should replace the file with the same name in the <ltib-install-directory>/dist/lfs-5.1/boa folder. Execute ltib again to build a new version of the Boa

web server. After the new version of Boa is built, replace the `boa.conf` file inside `<ltib-install-director>/rootfs/etc/conf` with the supplied `boa.conf` file.

The network interface options for LTIB are specified during the LTIB configuration and applied automatically by the startup scripts provided with LTIB. Therefore, no additional networking configuration is required unless your network settings are different. In that case, re-run `ltib` and change the default network settings to your preferences.

To get to the Boa test page, access the following URL using a browser: http://<board-ip>. The basic test web page should be displayed in the browser:

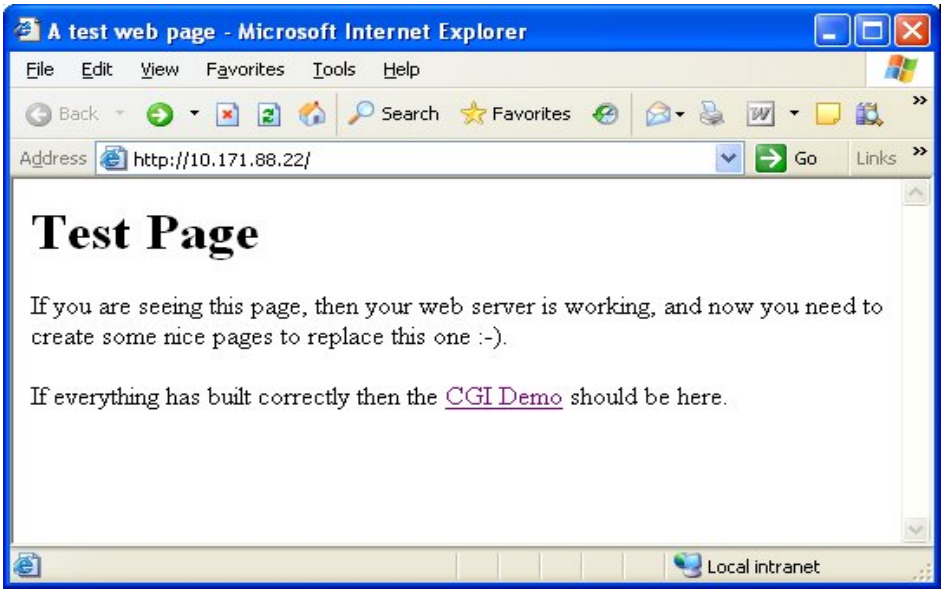

Figure 6. Boa Web Server Test Page

5 Developing CGI Extension for Boa WebServer

In this section we develop a basic application to demonstrate how to develop a program for uClinux running on the MCF5329EVB. The sample application is a CGI application. This means it talks to the web server using standard CGI protocol. The application displays an HTML form and sends the submitted messages to the CAN bus.

5.1 Application Structure

The next step is developing the CGI extension to send the messages to the CAN bus. The CGI extension displays the HTML form in the browser through which the message parameters to be sent to the CAN bus are specified. The CGI extension is developed using plain C.

The application contains the following source files:

Table 2. Application Source Files

Source File Name	Description
Makefile	Used by make tool to compile the application
can.c,can.h	C functions to access the CAN bus
Can4linux.h	Header file to access the Can4Linux CAN bus driver
cgi.c	main() function of the sample CGI application
cgivars.c, cgivars.h	Routines to get the CGI variables from the environment
htmllib.c, htmllib.h	Different functions to simplify the generation of the HTML output
template.c,template.h	Generation and processing of the HTML form to send CAN messages

The application sources are provided in Appendix F, “Source Files of the Example Application.” These are regular C programs. The CGI extensions are quite simple. All the parameters to them are passed in environment variables set by the web server and in standard input (in the case of a POST request). The standard environment variables with description are listed in the table below. The server re-sends the standard output to the client web browser.

Table 3. CGI Environment Variables

Environment Variable Name	Description
REQUEST_METHOD	HTTP request method used to invoke the CGI extension. Can be GET, HEAD, POST
QUERY_STRING	HTTP query (URL in browser with parameters)
CONTENT_LENGTH	Size of the data passed in stdin in the case of a POST HTTP request

5.2 Developing Application Using Open-Source Distribution

The source code for the uClinux distribution should be placed into the `uClinux-dist/user/cgican folder`.

The simplest way to add this application to the list of sources compiled in the uClinux distribution is by changing the following line in the `uClinux-dist/user/Makefile`:

```
dir_y += games
```

Should change to:

```
dir_y += games cgican
```

The content of the `Makefile` to build the application is as follows:

```
1:  EXEC = cgican
2:  OBJS = cgi.o cgivars.o htmllib.o template.o can.o
3:
```

```
4:  all: $(EXEC)
5:
6:  romfs:
7: $(ROMFSINST) $(ROOTDIR)/vendors/Generic/httpd /home/httpd
8:$(ROMFSINST) /home/httpd/cgi-bin/cgican
9:
10: $(EXEC): $(OBJS)
11:$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
12:
13:clean:
14:-rm -f $(EXEC) *.elf *.gdb *.o
15:
16:
17:$(OBJS): cgivars.h htmllib.h template.h
```

The first line defines the executable file name to be built. The second line defines the object files to be built executable. In general, from every C file one object file is built. The fourth line defines the default target. It is to build the executable file. The rules to install the built executable to the root-file system are defined from line 6 to line 9. The executable file is placed into the `/home/httpd/cgi-bin/cgican` file in the board's root-file system. `$(EXEC).clean rule` defines rules for the compilation and rules to clean up the built files. The top level uClinux `Makefile` defines all make variables like `CC, LDFLAGS, LDLIBS`. The last rule on line 17 defines the list of header files on which the object files depend. Due to this, the application is also rebuilt if any header file is changed.

5.3 Developing Application Using LTIB

Before building the application using LTIB, it must be first developed and packaged in `tar.gz`, `tar.bz2` or `srpm` archive/package. In our example, we use a `tar.bz2` archive.

The following `Makefile` can be used for the application:

```
1:  EXEC = cgican
2:  OBJS = cgi.o cgivars.o htmllib.o template.o can.o
3:
4:  all: $(EXEC)
5:
6: $(EXEC): $(OBJS)
7:$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
8:
9: clean:
10:-rm -f $(EXEC) *.elf *.gdb *.o
11:
12: $(OBJS): cgivars.h htmllib.h template.h
```

The first line contains the name of the executable file to be built. The second line contains all object files that should be built and linked to produce the final executable file. The fourth line defines the default rule — it is to build the output executable. Lines 6 to 7 define a standard rule to link object files. Lines 9 to 10 define a rule to clean the working folder. Line 12 defines the list of headers. The object files depend on these headers, so they are rebuilt if any are changed.

If, for example, such a Makefile is executed on a Linux PC, the `cgican` executable for it is built. LTIB generates the building environment, so instead of a normal GCC, the cross-compiled target GCC is executed, so under LTIB this `Makefile` produces the output for the selected target platform.

All the sources for LTIB must be packed by `tar` and `bzip2` into an archive. Call it `cgican-1.0.tar.bz2`. To create the archive the following commands can be used:

```
$ tar cvf * ../cgican-1.0.tar
$ bzip2 ../cgican-1.0.tar
```

LTIB uses RPM (Red Hat Package Manager) for its package management; therefore, the spec file for the RPM must be created to add a new package to LTIB. As a template, the `<install-dir>/dist/lfs-5.1/template/template.spec` file may be used.

The following spec file can be used for this CGI application:

```
1:  %define pfx /opt/freescale/rootfs/%{_target_cpu}
2:
3:  Summary         : CGI to send messages to CAN bus
4:  Name            : cgican
5:  Version         : 1.0
6:  Release         : 1
7:  License         : GPL
8:  Vendor          : Freescale
9:  Packager        : Freescale
10: Group           : Freescale
11: Source          : %{name}-%{version}.tar.bz2
12: BuildRoot       : %{_tmppath}/%{name}-%{version}
13: Prefix          : %{pfx}
14:
15: %Description
16: %{summary}
17:
18:%prep
19:
20:%setup
21:
22:%Build
23:make
24:
25:%Install
26:rm -rf $RPM_BUILD_ROOT
27:mkdir -p $RPM_BUILD_ROOT/%{pfx}/var/www/cgi-bin
28:install -m 777 cgican $RPM_BUILD_ROOT/%{pfx}/var/www/cgi-bin/
29:
30:%Clean
31:make clean
32:rm -rf $RPM_BUILD_ROOT
33:
34:%Files
35:%defattr(-,root,root)
36:%{pfx}/*
```

Lines 3 to 13 define information about the package, such as the author, license, etc. Lines 22 to 23 (section `%Build`) have commands to be executed to build the package. In our case, it is enough to run make. Lines 25 to 29 contain commands to install the built binary to the root-file system.

This spec file must be placed into the `<install-dir>/dist/lfs-5.1/cgican` directory and called `cgican.spec`. The package sources (`cgican-1.0.tar.bz2` archive) must be copied into the `/opt/freescale/pkgs` folder.

To test that the build is working, the following commands can be used:

Table 4. LTIB Command Lines

Command	Description
ltib –m prep –p cgican	Prepare the package to work on it. Unpack the sources into the `<install-dir>/rpm/BUILD/cgican-1.0` directory
ltib –m scbuild –p cgican	Build the package (run compilation)
ltib –m scinstall –p cgican	Run the installation sequence from the spec file
ltib –m scdeploy –p cgican	Install the building results into the target root-file system

Such commands can be used to work on any project and to change some projects. Only the project name needs changed (`-p` option). Also, LTIB allows capturing the changes in the form of the patch files. It then automatically updates the spec file to apply the patch after unpacking the package. To capture the changes in the form of a patch file, you can use the `ltib –m patchmerge –p <package>` command.

To add this package to the LTIB menu with the current packages, you must modify two additional files. First, add the following lines to the `<install-dir>/config/userspace/packages.lkc` file:

```
config PKG_CGICAN
bool "cgican"
help
 This package contains simple CGI extension to send messages to the CAN bus
```

Also, add the following line into the `<install-dir>/dist/lfs-5.1/common/pkg_map` file:

```
PKG_CGICAN = cgican
```

Add this line almost at the end of the above file:

```
# leave these as the last 2 packages in the build order
```

After this, the `cgican` package is added to the `Package list` menu of the LTIB. It is built and installed into the target root-file system if selected.

More documentation about LTIB can be found in the LTIB `<install-dir>/doc` directory.

5.4 Running the CGI Application

After uClinux on the board is running and the Boa web server is started, the sample application can be run. To invoke the application, you must use the `http://<board-ip>/cgi-bin/cgican` URL in the client web browser. Then, the form to specify parameters for the CAN message is displayed:

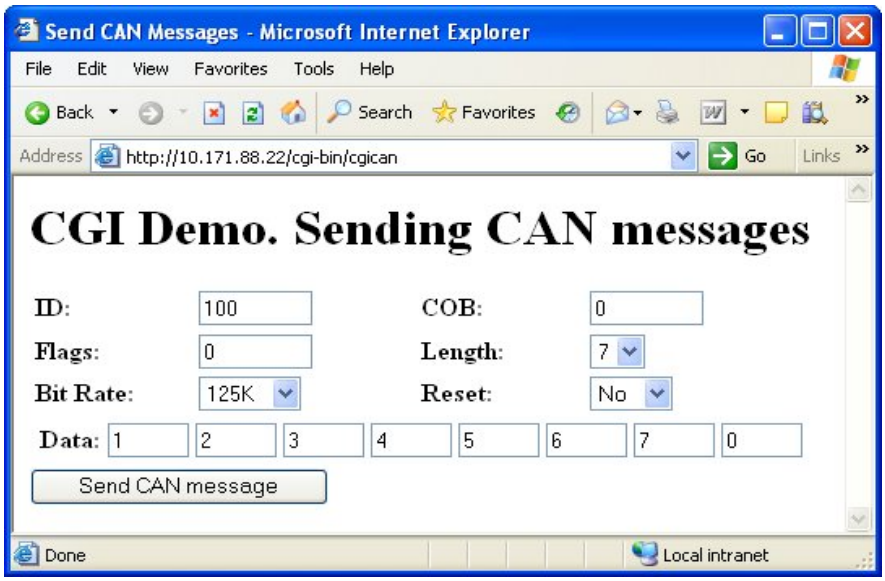

Figure 7. HTML Form of the Developed CGI Application

After specifying all data and pressing the `Send CAN Message` button, the message is sent to the CAN bus. It can be received, for example, by another board, another CAN device, or by CANalyzer software.

6 Conclusion

a) In this application note, building the sample CGI application was described. Building applications for uClinux is similar to building the application for a regular Linux PC; however, special considerations must be addressed such as using cross-compilers and code executed by an MPU without MMU. Installing and using the required Linux tools (compilers, source code, LTIB, etc.) was also described. This application note is a good starting point for how to use uClinux on the ColdFire MCF5329EVB. Using uClinux on all boards with ColdFire processors is similar; therefore, this information can also be used for other ColdFire devices/boards.

Appendix A Acronyms and Abbreviations

Table 5 lists acronyms and abbreviations used in this document

Table 5. Table of Acronyms

Term	Meaning
BSP	Board Support Package
CAN	Controller Area Network
CGI	Common Gateway Interface
CPU	Central Processing Unit
DHCP	Dynamic Host Resolution Protocol
FEC	Fast Ethernet Controller
LTIB	Linux Target Image Builder
MCU	Micro Controller Unit
MMU	Memory Management Unit
MPU	Micro Processor Unit
RAM	Random Access Memory
ROM	Read Only Memory
RPM	RedHat Package Manager
SCSI	Small Computer System Interface
USB	Universal Serial Bus

Appendix B dBUG Memory Map

Table 6. dBUG Memory Map

Type	Start	End	Port Size
flash (ext)	0x00000000	0x001FFFFF	16-bit
SDRAM	0x40000000	0x41FFFFFF	16-bit
SRAM (Int)	0x80000000	0x80003FFF	32-bit
dBUG code	0x00000000	0x0003FFFF	
dBUG data	0x40000000	0x4001FFFF	

Table 7. Chip Selects

Chip Select	Usage
CS0	Boot flash

Appendix C uClinux Memory Map

Table 8. uClinux Memory Map

Type	Start	End
SDRAM	0x40000000	0x41FFFFFF
NAND flash	0xD0000000	0xD10000000
Boot flash	0x00000000	0x001FFFFF[1]

1 The Linux kernel may be placed into boot flash and replace dBUG or be launched via dBUG (in this case Linux kernel should use memory above the dBUG reserved space — dBUG occupies first 0x40000 bytes of boot flash)

Table 9. uClinux Chip Selects

Chip Select	Usage
CS0	Boot flash
CS1	Register at address 0x10080000 (controls different signals on MCF5329EVB)
CS2	NAND flash

Appendix D Setting up the TFTP Server

To download the image to the board, the TFTP protocol is used; therefore, a TFTP server is required. It should be available in your Linux system. The following steps must be completed to configure the Linux host and the board:

- Disable any firewall software if you have it to allow the TFTP server to work (on Red Hat systems it can be done using `iptables -F` or run `setup`
- Enable the TFTP server in configuration of xinetd (on Red-Hat based systems). To do this create a file `/etc/xinet.d/tftp` with the following content:

```
{
disable = no
socket_type = dgram
protocol = udp
wait = yes
user = root
server = /usr/sbin/in.tftpd
server_args = /tftpboot
}
```

- Restart `xinetd`. To do this run the following command:

```
# /etc/init.d/xinetd restart
```

- Connect the board to the network
- Connect the board to the host via a serial cable
- Start `minicom` and configure it to talk to the board:
  - Serial setup: Select the correct serial port; Hardware & Software Flow control = No; Bps = 115,200
  - Modem & Dialing: Delete text for the following: Init String, Reset string, Hang-Up String, No Flow Control
  - Turn on auto wrap
- Power on the board. And you should see the console prompt for the dBUG ROM monitor flashed on to the board

On the board you should have dBUG v4a.1a.1c or later. Version 4b.1a.1d has extra features.

Appendix E Setting up the NFS Server

The following steps should be completed to allow this to happen:

- Install NFS server, if it is not installed
- Link root-file system to an exportable directory once you have built your project

```
$ ln –s <install-path>/ltib/rootfs /tftpboot/ltib
```

- Edit `/etc/exports` and add the following line there:

```
/tftpboot/ltib<target board IP>(rw, no_root_squash, async)
```

- Restart the NFS server. On Red Hat systems this could be done using the command:

```
$ /etc/init.d/nfsserver restart
Or
$ /sbin/service nfs restart
```

Appendix F Source Files of the Example Application

can.c

```
/* can.c */
/* Sending message to CAN bus using Can4Linux driver */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

#include "can.h"

/* Descriptor of CAN device */
static int can_fd = 0;

/* Opens CAN device */
int can_open()
{
    can_fd = open(CAN_DEVICE, O_RDWR);
    if (can_fd == -1) {
         return -1;
    }
    return 0;
}

/* Closes CAN device */
void can_close()
{
    if (can_fd>0) {
         close(can_fd);
         can_fd = 0;
    }
}

/* Reset CAN device */
void can_reset()
{
    volatile Command_par_t cmd;
    cmd.cmd = CMD_RESET;
    ioctl(can_fd, COMMAND, cmd);
}

/* Send the message to CAN bus */
int can_send_msg(canmsg_t* msg)
{
    if (can_fd>0) {
         write(can_fd, msg, 1);
        return 0;
    }
    return -1;
}

/* Sets the bit rate using IOCTL call */
```

```
int can_set_bitrate(int baud)
{
    Config_par_t cfg;
    volatile Command_par_t cmd;
    cmd.cmd = CMD_STOP;
    ioctl(can_fd, COMMAND, &cmd);
    cfg.target = CONF_TIMING;
    cfg.val1 = baud;
    ioctl(can_fd, CONFIG, &cfg);
    cmd.cmd = CMD_START;
    ioctl(can_fd, COMMAND, &cmd);
}
```

can.h file source

```
#ifndef _CAN_H_
#define _CAN_H_

#include "can4linux.h"

/* CAN device to be used */
#define CAN_DEVICE "/dev/can0"

int can_open();
void can_close();
void can_reset();
int can_send_msg(canmsg_t* msg);
int can_set_bitrate(int baud);


#endif
```

Can4linux.h

```
/*
 * can4linux.h - can4linux CAN driver module
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file "COPYING" in the main directory of this archive
 * for more details.
 *
 * Copyright (c) 2001 port GmbH Halle/Saale
 *-----------------------------------------------------------------
 * $Header: /cvs/sw/new-wave/user/can4linux/can4linux.h,v 1.1 2003/07/18 00:11:46 gerg Exp $
 *
 *-----------------------------------------------------------------------------
 *
 *
 * modification history
 * --------------------
 * $Log: can4linux.h,v $
 * Revision 1.1  2003/07/18 00:11:46  gerg
 * I followed as much rules as possible (I hope) and generated a patch for the
 * uClinux distribution. It contains an additional driver, the CAN driver, first
 * for an SJA1000 CAN controller:
 *   uClinux-dist/linux-2.4.x/drivers/char/can4linux
 * In the "user" section two entries
```

```
 *   uClinux-dist/user/can4linux     some very simple test examples
 *   uClinux-dist/user/horch         more sophisticated CAN analyzer example
 *
 * Patch submitted by Heinz-Juergen Oertel <oe@port.de>.
 *
 *
 *
 *
 *---------------------------------------------------------------------------
 */


#ifndef __CAN_H
#define __CAN_H


#ifndef __KERNEL__
#include <sys/time.h>
#endif
 /*---------- the can message structure */

#define CAN_MSG_LENGTH 8/**< maximum length of a CAN frame */


#define MSG_RTR   (1<<0)            /**< RTR Message */
#define MSG_OVR   (1<<1)            /**< CAN controller Msg overflow error */
#define MSG_EXT   (1<<2)            /**< extended message format */
#define MSG_PASSIVE(1<<4) /**< controller in error passive */
#define MSG_BUSOFF      (1<<5)/**< controller Bus Off  */
#define MSG_       (1<<6) /**<  */
#define MSG_BOVR(1<<7)    /**< receive/transmit buffer overflow */
/**
* mask used for detecting CAN errors in the canmsg_t flags field
*/
#define MSG_ERR_MASK(MSG_OVR + MSG_PASSIVE + MSG_BUSOFF + MSG_BOVR)

/**
* The CAN message structure.
* Used for all data transfers between the application and the driver
* using read() or write().
*/
typedef struct {
    /** flags, indicating or controlling special message properties */
    int             flags;
    int             cob; /**< CAN object number, used in Full CAN  */
    unsigned   long id; /**< CAN message ID, 4 bytes  */
    struct timeval  timestamp; /**< time stamp for received messages */
    short      int  length; /**< number of bytes in the CAN message */
    unsigned   char data[CAN_MSG_LENGTH]; /**< data, 0...8 bytes */
} canmsg_t;



/**
---------- IOCTL requests */

#define COMMAND  0/**< IOCTL command request */
```

```
#define CONFIG     1       /**< IOCTL configuration request */
#define SEND       2       /**< IOCTL request */
#define RECEIVE  3/**< IOCTL request */
#define CONFIGURERTR  4/**< IOCTL request */
#define STATUS           5      /**< IOCTL status request */

/*---------- CAN ioctl parameter types */

/**
 IOCTL Command request parameter structure */
typedef struct Command_par {
    int cmd;              /**< special driver command */
    int error;            /**< return value */
    unsigned long retval;/**< return value */
} Command_par_t ;


/**
 IOCTL Configuration request parameter structure */
typedef struct Config_par {
    int target;           /**< special configuration target */
    unsigned long val1;/**< 1. parameter for the target */
    unsigned long val2;/**< 2. parameter for the target */
    int error;            /**< return value for errno */
    unsigned long retval;/**< return value */
} Config_par_t ;

/**
 IOCTL CAN controller status request parameter structure */
typedef struct CanSja1000Status_par {
    unsigned int baud;    /**< actual bit rate */
    unsigned char status;/**< CAN controller status register */
    unsigned int error_warning_limit;/**< the error warning limit */
    unsigned int rx_errors;/**< content of RX error counter */
    unsigned int tx_errors;/**< content of TX error counter */
    unsigned int error_code;/**< content of error code register */
    unsigned int rx_buffer_size;/**< size of rx buffer  */
    unsigned int rx_buffer_used;/**< number of messages */
    unsigned int tx_buffer_size;/**< size of tx buffer  */
    unsigned int tx_buffer_used;/**< number of messages */
    unsigned long retval;/**< return value */
} CanSja1000Status_par_t;

/**
 IOCTL generic CAN controller status request parameter structure */
typedef struct CanStatusPar {
    unsigned int baud;                  /**< actual bit rate */
    unsigned long status;               /**< CAN controller status register */
    unsigned int error_warning_limit;   /**< the error warning limit */
    unsigned int rx_errors;             /**< content of RX error counter */
    unsigned int tx_errors;             /**< content of TX error counter */
    unsigned int error_code;            /**< content of error code register */
    unsigned int rx_buffer_size;        /**< size of rx buffer  */
    unsigned int rx_buffer_used;        /**< number of messages */
    unsigned int tx_buffer_size;        /**< size of tx buffer  */
    unsigned int tx_buffer_used;        /**< number of messages */
    unsigned long retval;               /**< return value */
```

```
    unsigned int type;                  /**< CAN controller / driver type */
} CanStatusPar_t;

/**
 IOCTL Send request parameter structure */
typedef struct Send_par {
    canmsg_t *Tx; /**< CAN message struct  */
    int error;            /**< return value for errno */
    unsigned long retval;/**< return value */
} Send_par_t ;

/**
 IOCTL Receive request parameter structure */
typedef struct Receive_par {
    canmsg_t *Rx; /**< CAN message struct  */
    int error;            /**< return value for errno */
    unsigned long retval;/**< return value */
} Receive_par_t ;

/**
IOCTL ConfigureRTR request parameter structure */
typedef struct ConfigureRTR_par {
    unsigned message;/**< CAN message ID */
    canmsg_t *Tx; /**< CAN message struct  */
    int error;            /**< return value for errno */
    unsigned long retval;/**< return value */
} ConfigureRTR_par_t ;

/**
---------- IOCTL Command subcommands */

# define CMD_START1
# define CMD_STOP 2
# define CMD_RESET3


/**
---------- IOCTL Configure targets */

# define CONF_ACC   0/* mask and code */
# define CONF_ACCM   1/* mask only */
# define CONF_ACCC   2/* code only */
# define CONF_TIMING3/* bit timing */
# define CONF_OMODE 4/* output control register */
# define CONF_FILTER5
# define CONF_FENABLE6
# define CONF_FDISABLE7

#endif   /* __CAN_H */
```

cgi.c

```
/* cgi.c */
/* main function of sample CGI application */
```

```
#include <stdio.h>
#include <string.h>

#include "cgivars.h"
#include "htmllib.h"
#include "template.h"


int main() {
    char **postvars = NULL; /* POST request data repository */
    char **getvars = NULL; /* GET request data repository */
    int form_method; /* POST = 1, GET = 0 */

      form_method = getRequestMethod();

      if(form_method == POST) {
           getvars = getGETvars();
           postvars = getPOSTvars();
      } else if(form_method == GET) {
           getvars = getGETvars();
      }

      htmlHeader("Send CAN Messages");
      htmlBody();

      /* display/process form */
      template_page(postvars, form_method);

      htmlFooter();
      cleanUp(form_method, getvars, postvars);

      fflush(stdout);
      exit(0);
}
```

cgivars.c

```
/* cgivars.c
 * (C) Copyright 2000, Moreton Bay (http://www.moretonbay.com).
 * see HTTP (www.w3.org) and RFC
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "cgivars.h"

/* local function prototypes */
char hex2char(char *hex);
void unescape_url(char *url);
char x2c(char *what);

/* hex2char */
/* RFC */
char hex2char(char *hex) {
         char char_value;
```

```
        char_value = (hex[0] >= 'A' ? ((hex[0] & 0xdf) - 'A') + 10 : (hex[0] - '0'));
        char_value *= 16;
        char_value += (hex[1] >= 'A' ? ((hex[1] & 0xdf) - 'A') + 10 : (hex[1] - '0'));
        return char_value;
}

/* unescape_url */
/* RFC */
void unescape_url(char *url) {
        int n, k;
        for(n=0, k=0;url[k];++n, ++k) {
                if((url[n] = url[k]) == '%') {
                        url[n] = hex2char(&url[k+1]);
                        k += 2;
                }
        }
        url[n] = '\0';
}


/* getRequestMethod
 * retn: from_method (GET or POST) on success,
 *                         -1 on failure.  */
int getRequestMethod() {
        char *request_method;
        int form_method;

        request_method = getenv("REQUEST_METHOD");
        if(request_method == NULL)
                return -1;

        if (!strcmp(request_method, "GET") || !strcmp(request_method, "HEAD") ) {
                form_method = GET;
        } else if (!strcmp(request_method, "POST")) {
                form_method = POST;
        } else {
                /* wtf was it then?!! */
                return -1;
        }
        return form_method;
}


/* getGETvars
 * retn: getvars */
char **getGETvars() {
        int i;
        char **getvars;
        char *getinput;
        char **pairlist;
        int paircount = 0;
        char *nvpair;
        char *eqpos;

        getinput = getenv("QUERY_STRING");
        if (getinput)
                getinput = strdup(getinput);
```

```
        /* Change all plusses back to spaces */
        for(i=0; getinput && getinput[i]; i++)
                if(getinput[i] == '+')
                        getinput[i] = ' ';

        pairlist = (char **) malloc(256*sizeof(char **));
        paircount = 0;
        nvpair = getinput ? strtok(getinput, "&") : NULL;
        while (nvpair) {
                pairlist[paircount++]= strdup(nvpair);
        if(!(paircount%256))
                        pairlist = (char **) realloc(pairlist,(paircount+256)*sizeof(char
**));
                nvpair = strtok(NULL, "&");
        }

        pairlist[paircount] = 0;
        getvars = (char **) malloc((paircount*2+1)*sizeof(char **));
        for (i= 0; i<paircount; i++) {
                if(eqpos=strchr(pairlist[i], '=')) {
                        *eqpos = '\0';
                unescape_url(getvars[i*2+1] = strdup(eqpos+1));
                } else {
                        unescape_url(getvars[i*2+1] = strdup(""));
        }
                unescape_url(getvars[i*2] = strdup(pairlist[i]));
        }
        getvars[paircount*2] = 0;
        for(i=0;pairlist[i];i++)
                free(pairlist[i]);
        free(pairlist);
        if (getinput)
                free(getinput);
        return getvars;
}


/* getPOSTvars
 * retn: postvars */
char **getPOSTvars() {
        int i;
        int content_length;
        char **postvars;
        char *postinput;
        char **pairlist;
        int paircount = 0;
        char *nvpair;
        char *eqpos;

        postinput = getenv("CONTENT_LENGTH");
        if (!postinput)
                exit(1);
        if(!(content_length = atoi(postinput)))
                exit(1);
        if(!(postinput = (char *) malloc(content_length+1)))
                exit(1);
```

```
        if (!fread(postinput, content_length, 1, stdin))
                exit(1);
        postinput[content_length] = '\0';


        for(i=0;postinput[i];i++)
                if(postinput[i] == '+')
                        postinput[i] = ' ';

        pairlist = (char **) malloc(256*sizeof(char **));
        paircount = 0;
        nvpair = strtok(postinput, "&");
        while (nvpair) {
                pairlist[paircount++] = strdup(nvpair);
                if(!(paircount%256))
                        pairlist = (char **) realloc(pairlist, (paircount+256)*sizeof(char
**));
                nvpair = strtok(NULL, "&");
        }

        pairlist[paircount] = 0;
        postvars = (char **) malloc((paircount*2+1)*sizeof(char **));
        for(i = 0;i<paircount;i++) {
        if(eqpos = strchr(pairlist[i], '=')) {
                        *eqpos= '\0';
                unescape_url(postvars[i*2+1] = strdup(eqpos+1));
        } else {
                        unescape_url(postvars[i*2+1] = strdup(""));
                }
        unescape_url(postvars[i*2]= strdup(pairlist[i]));
        }
        postvars[paircount*2] = 0;

        for(i=0;pairlist[i];i++)
                free(pairlist[i]);
        free(pairlist);
        free(postinput);

        return postvars;
}

/* cleanUp
 * free the mallocs */
int cleanUp(int form_method, char **getvars, char **postvars) {
        int i;

        if (postvars) {
                for(i=0;postvars[i];i++)
                        free(postvars[i]);
                free(postvars);
        }
        if (getvars) {
                for(i=0;getvars[i];i++)
                        free(getvars[i]);
                free(getvars);
        }
```

```
        return 0;
}
```

cgivars.h

```
/* cgivars.h */

#ifndef _CGIVARS_H
#define _CGIVARS_H

/* method */
#define GET0
#define POST1


/* function prototypes */
int getRequestMethod();
char **getGETvars();
char **getPOSTvars();
int cleanUp(int form_method, char **getvars, char **postvars);


#endif   /* !_CGIVARS_H */
```

htmllib.c

```
/* htmllib.c
 * HTML common library functions for the CGI programs. */

#include <stdio.h>
#include <string.h>
#include "htmllib.h"


void htmlHeader(char *title) {
  printf("Content-type: text/html\n\n<html><head><title>%s</title></head>",
                   title);
}

void htmlBody() {
    printf("<body>");
}

void htmlFooter() {
    printf("</body></html>");
}

void addTitleElement(char *title) {
        printf("<h1>%s</h1>", title);
}

void printInputTextInt(char* name, char* var, int default_value) {
    char buffer[20];
    snprintf(buffer, 20, "%d", default_value);
    printInputText(name,var,buffer);
```

```
}

void printInputText(char* name, char* var, char* default_value) {
        printf("<td><b>%s:</b></td>\n", name);
        printf("<td><input type='text' name='%s' style='width: 70px;' value='%s'/></td>\n",
var, default_value);
}

void printInputSelect(char* name, char* var, char* default_value, char** values) {
        printf("<td><b>%s:</b></td>\n", name);
        printf("<td><select name='%s'>\n", var);
        if (values!=NULL) {
                char* value;
                int i=0;
                while ((value=values[i])!=NULL) {
                        printf("<option value='%s'", value);
                        if (strcmp(value,default_value)==0) {
                            printf(" selected");
                        }
                        printf(">");
                        ++i;
                        value = values[i];
                        if (value!=NULL) {
                                printf("%s", value);
                                ++i;
                        }
                        printf("</option>\n");
                }
        }
        printf("</select></td>\n");
}

void printError(char* msg)
{
    printf("<br><font color='red'>%s</font><br>", msg);
}
```

htmllib.h

```
/* htmllib.h */

#ifndef _HTMLLIB_H
#define _HTMLLIB_H

/* function prototypes */
void htmlHeader(char *title);
void htmlBody();
void htmlFooter();
void addTitleElement(char *title);
void printInputSelect(char* name, char* var, char* default_value, char** values);
void printInputText(char* name, char* var,char* default_value);
void printInputTextInt(char* name, char* var,int default_value);
void printError(char* msg);

#endif   /* !_HTMLLIB_H*/
```

template.c

```
/* template.c */
/* Displays and process HTML form */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "cgivars.h"
#include "htmllib.h"
#include "can4linux.h"
#include "can.h"

/* Form submit URL */
#define CGI_APPLICATION_URL "/cgi-bin/cgican"

#define ID_VAR "id"
#define COB_VAR "cob"
#define FLAGS_VAR "flags"
#define LENGTH_VAR "length"
#define BITRATE_VAR "bitrate"
#define CANRESET_VAR "canreset"
#define DATA_VAR "data"

/* Value/display pairs for Length HTML select control */
char* length_values[]={
                       "0", "0",
                       "1", "1",
                       "2", "2",
                       "3", "3",
                       "4", "4",
                       "5", "5",
                       "6", "6",
                       "7", "7",
                       NULL};

/* Value/display pairs for Bit Rate HTML select control */
char* bitrate_values[]={
                       "20", "20K",
                       "50", "50K",
                       "100", "100K",
                       "125", "125K",
                       "250", "250K",
                       "500", "500K",
                       "800", "800K",
                       "1000", "1000K",
                       NULL};

/* Value/Display pairs for Reset HTML select control */
char* reset_values[]={
                       "0", "No",
                       "1", "Yes",
                       NULL};


/* Display HTML form */
```

```
int template_page(char **postvars, int form_method) {
        int i;
        canmsg_t message;
        int bitrate=125;
        int canreset=0;
        char buffer[20];

        /* Default CAN message parameters */
        message.id = 100;
        message.flags = 0;
        message.cob = 0;
        message.length = 7;
        for (i=0; i<CAN_MSG_LENGTH; i++)
            message.data[i] = 0;


        addTitleElement("CGI Demo. Sending CAN messages");

        /* Get the variables values from form if it is submitted */
        if(form_method == POST) {
                for (i=0; postvars[i]; i+=2) {
                    if (strcmp(postvars[i], ID_VAR)==0) {
                         message.id = atoi(postvars[i+1]);
                    } else if (strcmp(postvars[i], COB_VAR)==0) {
                         message.cob = atoi(postvars[i+1]);
                    } else if (strcmp(postvars[i], FLAGS_VAR)==0) {
                         message.flags = atoi(postvars[i+1]);
                    } else if (strcmp(postvars[i], LENGTH_VAR)==0) {
                         message.length = atoi(postvars[i+1]);
                    } else if (strcmp(postvars[i], BITRATE_VAR)==0) {
                         bitrate = atoi(postvars[i+1]);
                    } else if (strcmp(postvars[i], CANRESET_VAR)==0) {
                         canreset = atoi(postvars[i+1]);
                    } else if (strncmp(postvars[i], DATA_VAR, strlen(DATA_VAR))==0) {
                         int data_index;
                         data_index = atoi(postvars[i]+strlen(DATA_VAR));
                         if (data_index>=0 && data_index<8) {
                             message.data[data_index] = atoi(postvars[i+1]);
                         }
                    }
                }
                /* Send CAN message */
                if (can_open()!=-1) {
                    if (canreset) {
                         can_reset();
                    }
                    can_set_bitrate(bitrate);
                    can_send_msg(&message);
                    can_close();
                } else {
                    printError("CANNOT OPEN CAN DEVICE!");
                    perror("CAN");
                    printf("<br>");
                }

        }
```

```
        /* Output form */
        printf("<form action=\"%s\" METHOD=POST>\n", CGI_APPLICATION_URL);
        printf("<table border='0'>\n");
        printf("<tr>\n");
        printInputTextInt("ID", ID_VAR, (int)message.id);
        printInputTextInt("COB", COB_VAR, message.cob);
        printf("</tr>\n");
        printf("<tr>\n");
        printInputTextInt("Flags", FLAGS_VAR, message.flags);
        snprintf(buffer, 20, "%d", message.length);
        printInputSelect("Length", LENGTH_VAR, buffer, length_values);
        printf("</tr>\n");
        printf("<tr>\n");
        snprintf(buffer, 20, "%d", bitrate);
        printInputSelect("Bit Rate", BITRATE_VAR, buffer, bitrate_values);
        snprintf(buffer, 20, "%d", canreset);
        printInputSelect("Reset", CANRESET_VAR, buffer, reset_values);
        printf("</tr>\n");
        printf("<tr><td colspan='4'><table><tr>\n");
        printf("<td><b>Data:</b></td>\n");
        for (i=0;i<8;i++) {
                printf("<td><input type='text' name='%s%d' style='width: 50px;'
value='%d'/></td>\n",
                        DATA_VAR, i, (int)message.data[i]);
        }
        printf("</tr></table></td></tr>\n");
        printf("</table>\n");
        printf("<input type='submit' value='Send CAN message'/>\n");
        printf("</form>\n");
        return 0;
}
```

template.h

```
/* template.h */
#ifndef _TEMPLATE_H
#define _TEMPLATE_H

int template_page(char **postvars, int form_method);

#endif
```

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3408
Rev. 0
02/2007

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© Freescale Semiconductor, Inc. 2006. All rights reserved.

freescale™
semiconductor