C Data

Solutions

Building uClinux

for the

Compact Flash Computer

Version 1.0

24 Nov 2004

Notice

The information in this document is subject to change without notice.

THE SOFTWARE ANF DOCUMENTION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, C Data Solutions Ltd DOES NOT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING USE, OR THE RESULT OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

uClinux distribution

Installing the LittleBoy distribution

The LittleBoy distribution CD contains all the tools and sources needed to start developing uClinux applications. If you haven t already a m68k-elf toolchain installed on your machine you can do so now by either using the supplied Makefile on the CD or directly executing the installer script.

The toolchain should be installed in the location /usr/local and the distribution in a folder called littleboy in the home directory of the user you we re issuing this command.

You can install the toolchain manually by using the command:

```
sh $path_to_cdrom/m68k-elf-tools-20030314.sh
```

Similar for the uclinux distribution you can manually untar the files by using following command:

```
tar xzvf $path_to_cdrom/uclinux.tar.gz
```

This will install the distribution in a folder uclinux in the current location.

Development Source Tree

To untar the package, type:

```
tar xzvf $path_to_cdrom/uclinux.tar.gz
```

The directory will contain all the sources that are needed to compile the target image with our options. The source tree has the following structure:

bin	Utilities for uClinux platform to create the flash.bin.
Documentation	Some details of the uClinux/ColdFire.
tools	uClinux/ColdFire tools for compiler and other build tools.
user	User applications and sources for them.
freeswan	A free IPSec program providing security functions, authentication and encryption. It requires some additional libraries. glibc Replacement libraries.
lib	All the application libraries.
linux-2.x.x	Patched uClinux kernel sources
uClibc	New libc version.
config	Configuration for setting up the uClinux file system. It is used to drive the vendor configuration.
romfs	ROM based file system structure. It includes user application binaries and device nodes. This folder is created after the compilation.
vendors	Vendor specific build instructions. The folder includes subdirectories for all the support platforms.
images	After the building process this folder will include the final build binaries of the kernel, ROM file system and the companied image.

The directories in the romfs folder contains the same tree that will eventually be created to the target machine The folders will also contain the configurations files. The file structure is basically the same as in the standard Linux:

bin	System program binaries.
dev	Files that provide an interface to a physical device.
etc	System configuration. This is where unchangeable configuration information is stored
home	User home directories.
lib	Includes the shared libraries, when supported.
mnt	The mount points.
proc	The mount point stub of the virtual proc file system.

tmp	For temporary files. Normally linked to /var/tmp
usr	Additional utilities and applications
var	The variable files. Normally resides in a RAM filesystem.

The LittleBoy system uses a special kernel code which maps the bootloader environment into individual proc-filesystem entries. So it is possible to change IP settings and other information inside the bootloader without building a new uclinux image for changes to take affect.

For example to use the settings inside the bootloader for network settings you can use it inside your startup script as:

```
hostname `cat /proc/config/hostname`
ifconfig lo 127.0.0.1
route add -net 127.0.0.0
netmask 255.0.0.0 lo
ifconfig eth0 `cat /proc/config/ipaddr` netmask `cat /proc/config/netmask`
route add default gw `cat /proc/config/gatewayip`
```

Compiling the uClinux Kernel

The kernel is the core of the operating system. It takes care of all process management (what program runs and when), memory management (which parts of memory get used for what) and also, the kernel takes care of interfacing the OS with the hardware.

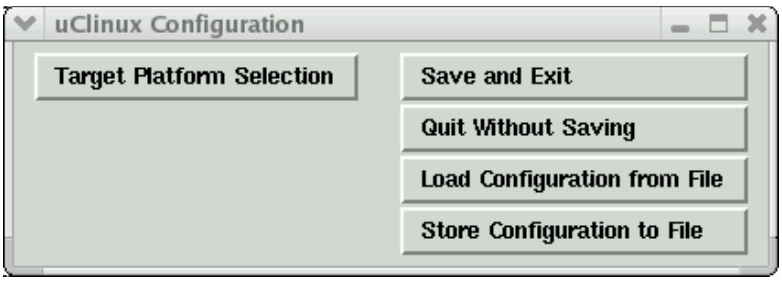
Before you compile your kernel, you need to configure it. Configuration is your opportunity to control exactly what kernel features are enabled or disabled in your new kernel. You will also be in control of what parts get compiled into the kernel binary image (which gets loaded at boottime). Before starting the configuration process you must be aware what user applications you need, because also these are compiled to form the image.

Configuration to uClinux kernel is done by making a configuration script that eventually builds the kernel. This can be executed with the command **make config**. The make config command needs bash to work. The bash is a command language interpreter that can execute commands from a file. Using **make config** command is quite a laborious and old-fashioned way to configure uClinux. Nowadays there are other methods to do the configuration in a more user friendly way. These alternatives include following commands:

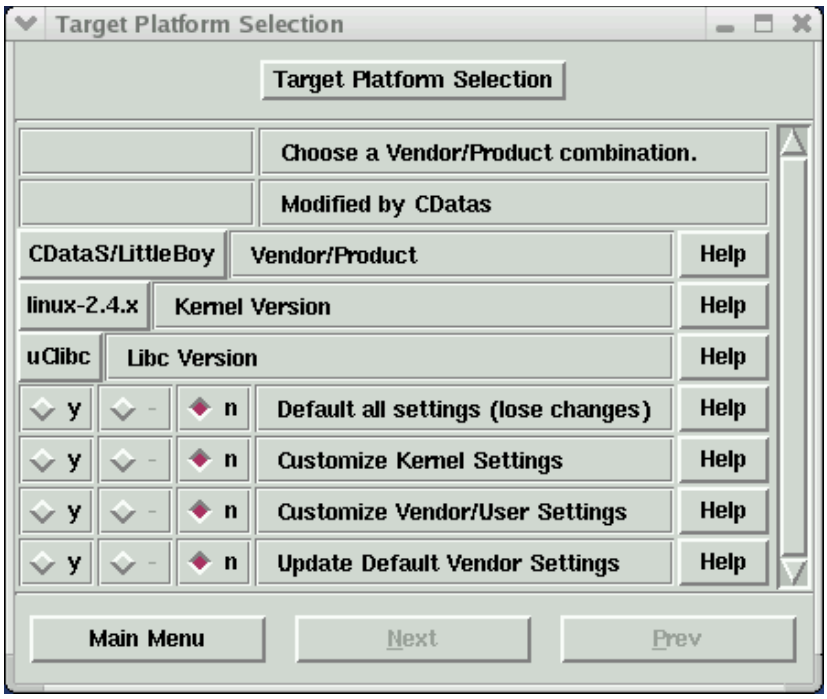
make menuconfig	This is a text based coloured configuration with a possibility to mark options
make xconfig	This is an X window based option for configuration
make oldconfig	This command is useful if you want to carry your existing configuration to a new version. Here only non-existing configurations are asked. Old configurations are red from the ./.config file

All of these options will lead to the same result. The choices done by the user will affect the size of the image. This depends from the drivers and support functions that are included.

The uClinux configuration and build process is highly similar to the standard Linux **make xconfig** process. The only major difference is that in the configuration phase the user is also able to decide which applications are compiled into the final image. Executing the xconfig will prompt the following screen:



Select 'Target Platform Selection”, this will prompt the following screen:



Under the Target **Platform Selection** the user can choose whether to customize the kernel settings and user applications. The user is also able to reset the default settings and choose between two C libraries but should note that some of the LittleBoy specific applications are configured to use the newer uClibc. After the choices have been made the **Save and Exit** from the Main Menu will continue to the next configuration screens.

The default settings for the LittleBoy will create a fully working image and configuration of the image is basically necessary only when adding and removing the user applications.

Once your kernel and user applications are configured, it s time to get it compiled. Before we can compile it, we need to generate dependency information. The dependency information can be built with the **make dep** command, which checks each C file in the source and figures out which header file it depends on.

The user-defined kernel is now ready to compile. To make the binary image simply type make command. As the kernel is being made, the display should show all of the necessary (or dependant) files being compiled and linked together into one large binary file which is saved to the /images directory of the uClinux tree by the make file. This file can then be downloaded onto the target board. The default build generates a raw binary image (*image.bin*) containing both the uClinux kernel and the ROM filesystem, which is a space-efficient, small, read-only filesystem.

The procedure for compiling the kernel described above can be combined to following steps:

1. Configure the build process using an option of your choice: make menuconfig or make config or make xconfig.
1. 2. Build the dependencies by typing make dep.

2. 3. Build the image and source tree by typing make.

Some Notes on Make Clean and other Make Commands

The uClinux provides a mechanism to clean up the system from unnecessary object files and from other created folders and files. The simple version of the cleaning methods is the **make clean** command execution. This will remove the folders ROMFSDIR and IMAGDIR and also erase the config.tk if xconfig has been used and all the old entries from the previous build. More thorough option is the make **mrproper,** which will clean all the created configuration files. After this command the system will be in its original state and all the configurations will be needed to do again.

uClinux environment also provides some useful make commands that can be used to compile certain parts of the distribution. These commands are:

make user_only	Scouts through the user tree and do what it needs.
make romfs	Makes only the ROM file system.
make image	Runs genromfs and creates images.
make linux	Compiles the Linux kernel.
make lib_only	Compiles only the libraries.
make user_clean	Cleans up the user folder by removing all the object.

Porting Applications

Porting applications to the uClinux is a similar task than in the standard Linux. Because Linux already has lots of applications developed under the General Public License (GPL), they can be added with minor modifications to run under the uClinux kernel. The first step for porting applications is to modify the applications Makefiles and configuration files to ensure that the compilation is done with the uClinux tool chain. Furthermore, the lack of the MMU and related system calls should be noted.

Most of the changes are made to the architecture of memory management subsystem, that has been modified by removing the parts that refer to the MMU hardware. This includes the modification of the subsystems by rewriting and removing the parts

interacting the MMU unit. All though this seems like a large modification, the kernel and the user space software hacking is rather small. The lack of memory management and memory protection are important issues when implementing new code under the uClinux platform.

When compiling the newly ported application, most of the problems are generated from the libraries. The lack of the library forces the user to port the whole library to the system. Then when linking the code with these libraries some unexpected problems might occur. Also the Application Programming Interfaces (APIs) offered by the C libraries may cause some differences. Of the C libraries for developing embedded Linux systems the uClibc is much smaller than the glibc (GNU C Library), but nearly all applications supported by glibc also work perfectly with uClibc. Porting applications from glibc to uClibc typically involves just recompiling the source code. uClibc even supports shared libraries and threading has a good support for Linux like API s.

Important Notes on the Changes in Programming Interfaces

The lack of MMU in uClinux environment is the biggest change compared to the standard Linux. This causes the lack of memory protection and a virtual memory model. This of course affects the development done under the uClinux by forcing some changes to be adopted to some system calls and the fact that the programmer is responsible for memory management.

The **fork()** system call is used to duplicate the current process by creating a new entry in the process table. This can be handy if the program handles more than one function at a time. The created child process is almost identical to the parent executing the same code but with its own data space, environment and descriptors. The fork command is implemented by using copy-onwrite pages. When either one of the process tries to write on the page frame, a private copy of the page is created for this process. The new physical page is mapped into the original logical address space. Without MMU the process cannot be completely and securely clone a process, nor does it have access to copy-on-write page.

The uClinux implements BSDs **vfork()** in order the offer the functionality of fork. The process created by this system call shares all their memory space including the stack. To prevent the parent to override the data needed by the child process the parent is suspended until the child exists.

Memory allocation (**malloc)** is normally implemented at the low level using the **brk** system call. Under Linux environment each created process owns a specific memory region called a heap. This is used to for processes dynamic memory request. The size of the memory region can be increased or decreased directly by the **brk()** system call. Under uClinux this system call cannot increase the memory region unless it goes through some changes.

The process is allowed to allocate memory from the global pool of free memory. The simplest memory allocation method under uClinux is mmap() and munmap() to return the memory to the memory pool. The choice of different methods for memory allocation

depends also from the libc version. Both uC-libc and uClibc support a simple memory allocator, **malloc-simple**, which uses the **mmap** and **munmap** that lets the kernel actually to handle the requests for memory. The problem with this call is that the based allocation is about 56 bytes, which can cause small memory request to grow quite high. The main problem lies on the fact that efficient and fast memory allocation generally also increases code size to small application.

Step by Step Procedure for Adding User Application

The following gives simple instructions for adding a user-written application to the uClinux con- figuration system. Entries must be added to three files, and an appropriate Makefile must exist in the user application source directory, which must be put in user (all directory names here are given relative to the uClinux top directory).

First you need to modify *user/Makefile.* You need to add the new application directory to the list of directories to be built. Using the convention **CONFIG_<user dir>_<application dir>_<application binary>**, for the foo example we would add:

```
DIR_$(CONFIG_USER_FOO_FOO) += foo
```

The next file you need to modify is *config/Configure.help.* This file contains the text which is presented on request during the config. Add a block like:

```
CONFIG_USER_FOO_FOO
  This program does fooey things to your bars.
```

The text must be intended two spaces, and there must be no empty lines. Lines should be <70 characters long.

The next file you need to modify is config/config.in. Add a line in the appropriate menu section (that is in the program group you want your application to show up in during **make config.** You can use e.g. misc.), like

```
bool   foo   CONFIG_USER_FOO_FOO
```

Next there needs to be a proper /user/foo/Makefile . The Makefile should follow

```
EXEC = foo
OBJS = foo.o
all: $(EXEC)
$(EXEC): $(OBJS)
     $(CC) $(LDFLAGS)  o $@ $(OBJS) $(LDLIBS)
romfs: $(ROMFSINST) /bin/$(EXEC)
clean: rm romfs:
```

If more than one executable is built in the foo directory, as above, then the Makefile should look like

```
EXECS = foo bar
OBJS = foo.o bar.o
all: $(EXECS)
$(EXECS): $(OBJS)
     $(CC) $(LDFLAGS)  o $@ $@.o $(LDLIBS)
romfs:
     $(ROMFSINST)  e CONFIG_USER_FOO_FOO /bin/foo
     $(ROMFSINST)  e CONFIG_USER_FOO_BAR /bin/bar
```

More complex makefiles are of course possible. After all this is set up, doing the standard build process: (**make xconfig**; **make dep**; **make)** should build the application and install it in *romfs* and hence in the target system *image.bin*.

FLASHING UCLINUX

To FLASH the uClinux image the following steps should be followed.

1. From ppcboot load image.pgk into ram
2. eraes the target FLASH 0xff840000-0xff91ffff
3. copy the image from ram to FLASH

LOAD UCLINUX TO RAM

```
LittleBoy>tftp 20000 image.pkg
00:c0:1b:05:fa:2d
probing slot 0
probing slot 1
probing slot 2
probing slot 3
probing slot 4
probing slot 5
probing slot 6
probing slot 7 00:C0:1B:05:FA:2D *
ARP broadcast 1
TFTP from server 192.0.0.3; our IP address is
192.0.0.10
Filename 'image.pkg'.
Load address: 0x20000
```

```
Loading:
#################################################
#################################################
#########################################
done
Bytes transferred = 743266 (b5762 hex)
Automatic start not enabled...
```

ERASE TARGET FLASH

```
LittleBoy>erase ff840000 ff91ffff
Erase Flash from 0xff840000 to 0xff91ffff
Erasing sector 11 ... ok.
Erasing sector 12 ... ok.
Erasing sector 13 ... ok.
Erasing sector 14 ... ok.
Erasing sector 15 ... ok.
Erasing sector 16 ... ok.
Erasing sector 17 ... ok.
Erasing sector 18 ... ok.
Erasing sector 19 ... ok.
Erasing sector 20 ... ok.
Erasing sector 21 ... ok.
Erasing sector 22 ... ok.
Erasing sector 23 ... ok.
Erasing sector 24 ... ok.
Erased 14 sectors
LittleBoy>
```

COPY IMAGE TO FLASH

```
LittleBoy>echo $(filesize)
b5762
LittleBoy>cp.b 20000 ff840000 $(filesize)
Copy to Flash... done
LittleBoy>
```