Embedded uClinux, the Altera DE2, and the SHIM Compiler

Wei-Chung Hsu, David Lariviere, and Stephen A. Edwards
Columbia University, Department of Computer Science
wh2138@cs.columbia.edu, dal2103@columbia.edu, sedwards@cs.columbia.edu

Abstract

SHIM is a concurrent deterministic language focused on embedded system. Although SHIM has undergone substantial evolution, it currently does not have a code generator for a true embedded environment.

In this project, we built an embedded environment that we intend to use as a target for the SHIM compiler. We add the uClinux operating system between hardware devices and software programs. Our long-term goal is to have the SHIM compiler generate both user-space and kernel/module programs for this environment. This project is a first step: we manually explored what sort of code we ultimately want the SHIM compiler to produce.

In this report, we provide instructions on how to build and install uClinux into an Altera DE2 board and example programs, including a user-space program, a kernel module, and a simple device driver for the buttons on the DE2 board that includes an interrupt handler.

1 Introduction

The idea of this project is to provide an embedded system target for the SHIM compiler and ultimately for use within the CSEE 4840 Design of Embedded Systems course at Columbia University. This is part of longer-term project whose goal is to produce a compiler for the SHIM language able to target this platform.

SHIM, Software/Hardware Integration Medium, is a concurrent deterministic language. Its user-specified parallelism is designed to map naturally to mixed hardware/software systems and allow programmers to take advantage of their inherent parallelism.

One dimensional along which to characterize embedded systems is the quantity of resources at their disposal. Embedded systems range from incredibly resource-constrained four-bit microcontroller-based systems all the way to systems with more compute power than a typical desktop computer. Above a certain point in this spectrum, virtually every system has sufficient resources to include a formal operating system. It is these systems that we are concerned with here.

For this project, we are using the Altera DE2 board as a target. This board has a Cyclone II EP2C35F672C6 FPGA, 8M SDRAM, 512K SRAM and 4M flash, and is sufficiently powerful to support a general-purpose operating system.

After looking at what is available, we decided to use the port of uClinux, a Linux variant designed to work in environments without memory-management units, as our operating system. uClinux has already been ported to the DE2 board, is open source, and Linux is of growing importance in the embedded systems arena.

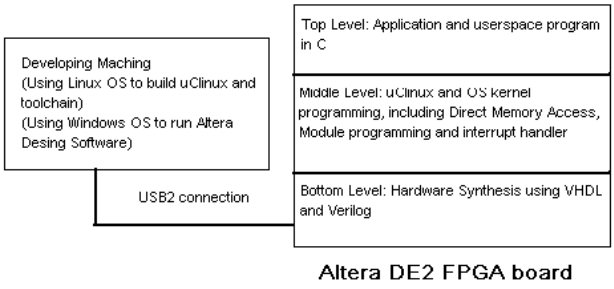


Figure 1: System Architecture

Embedded operating systems are mature and are found in most embedded systems. Many operating systems suitable for embedded systems are available, including WinCE, various Linux variants, Wind River's VxWorks, eCos, RTEMS, FreeRTOS, μC/OS-II, and many others. WinCE is resource hungry, usually requiring at least 32M of RAM and a 100 MHz processor.

Linux variants are open source and more malleable. They can also have much smaller footprints, as little as a 1 MB image that requires less than 4 MB of RAM. Because of our resource constraints and desire to use an open-source OS, we chose uClinux.

2 Objectives and System Architecture

The current SHIM compiler produces single-threaded C code suitable for running on a desktop workstation. Our long-term goal is a SHIM compiler able to produce code suitable for execution on embedded platforms. The goal of this project is to develop and understand such a platform.

2.1 System Architecture

There are three layers in this embedded system and a development environment. (Figure 1). The bottom layer is the hardware: the Altera DE2 FPGA board and its programmable hardware. To simplify our lives, we used the configuration provided by Altera with the DE2 board. The middle layer is uClinux and the top layer is the application—the user-space program. Our project focuses on middle and

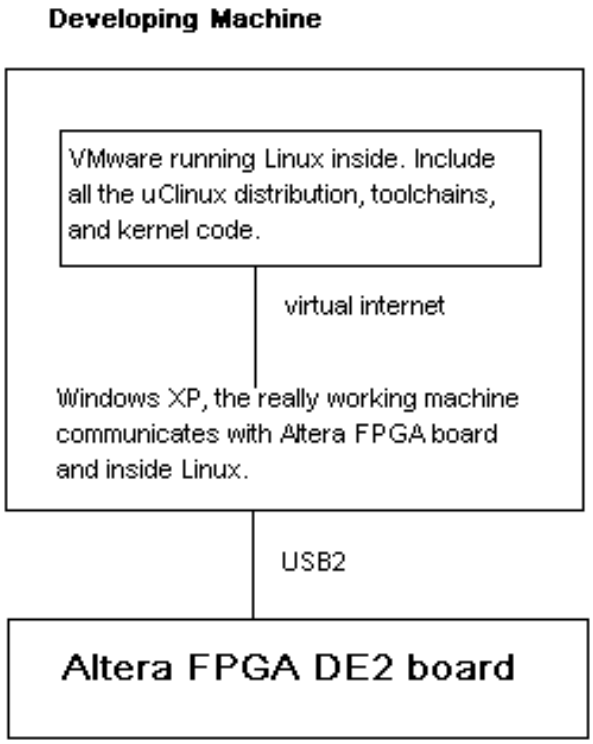


Figure 2: Development Machine Architecture

top layers. In addition to the target machine, a development machine connects to the DE2 board through USB to upload the code image to the board and control it.

2.2 *Development Machine*

We ran the Altera software on Windows, but the uClinux distribution, kernel and toolchains all run under Linux, even the code image and object file format is the Linux FLAT format. A straightforward approach would be to use a separate Linux box as the uClinux development platform, but to reduce the number of computers we had to use, we chose instead to use a VMware virtual machine to run the uClinux development platform.

Figure 2 illustrates this configuration: Windows XP with Linux running on a VMware inside. A virtual network connects the Windows and Linux environment. Figure 3 is a screenshot of us working in Windows with with Linux and the Altera DE2. We use *ssh* to build uClinux and download the image to the Windows side. Then we use the Altera software to synthesize hardware and load the image into FPGA. Below, we provide details about how to build uClinux and use the Altera software.

2.3 *Low-level Hardware*

To simplify our work, we used the sample code and configuration provided by Altera with the DE2 board. This can be downloaded from `ftp://ftp.altera.com/up/pub/de2/DE2_System_v1.5.zip`. This file contains all the Ver-

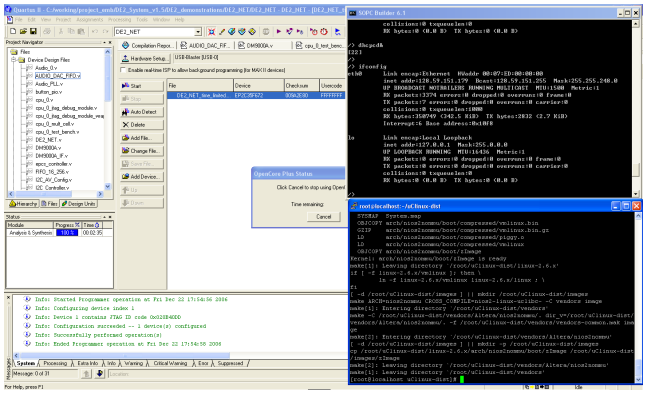


Figure 3: Working environment example

ilog and VHDL sample code in the DE2_demonstrations folder. For this project, we used the DE2_NET configuration.

2.4 *The Middle level: uClinux*

This is the focus of our project. We choose uClinux as our embedded OS. It is a derivative of the Linux 2.0 kernel intended for microcontrollers without memory management units. We used the uClinux-dist-20060803 version with the linux-2.6.17-uc1 kernel.

Since there is no MMU in a uClinux system, a user-space program has direct access to the hardware and can therefore bypass most kernel code. However, there is still a need for kernel modules since they appear to be the only way to register an interrupt handler. Of course, we could opt to use polling exclusively, but this is would be inefficient.

2.5 *The Top Level: Application programs*

As we mentioned above, since there is no MMU, a user-level program can directly access the hardware if it has its addresses. These are defined in the uClinux-dist/linux-2.6.x/include/nios2_system.h header file.

We also tried some standard applications, such as telnetd, insmod, ftpd, etc.

We wrote several programs to test the entire system at each level and tried to see what functionality uClinux provides. We found that each new version of uClinux supports more functionality and has more comprehensive hardware support.

3 Building uClinux and Its Toolchain

Here, we describe how to install the uClinux toolchain and use it to build the kernel image. Running the system ultimately requires the Altera software, Quartus II v6.1 and the Nios II Embedded Design Suite v6.1, because the sample code DE2_System_v1.5 works with v6.1. Since the Altera has detail documentation on how to use their design software, we will not discuss it.

Download the uClinux distribution version 20060803 from `http://www.uclinux.org/pub/uClinux/dist/uClinux-dist-20060803.tar.bz2` and

toolchains from `http://nioswiki.jot.com/WikiHome/OperatingSystems/%C2%B5Clinux/BinaryToolchain/nios2gcc.tar.bz2`. The Nios II Wiki (`http://en.wikipedia.org/wiki/Nios_II`) taught us to build uClinux and how to write modules, interrupt handlers, and user-space programs.

3.1 Install Toolchain First

Building the uClinux kernel requires the toolchain be installed. This includes the gcc cross-compiler. As root, extract the toolchain by typing

```
tar jxf nios2gcc.tar.bz2 -C /
```

This places the cross gcc tools in /opt/nios2. Set up the PATH for the cross gcc. This can be done by adding it to ~/.bash_profile, i.e., `PATH=$PATH:/opt/nios2/bin:$HOME/bin`, or just from the command-line with `PATH=$PATH:/opt/nios2/bin`. To verify the cross gcc, try

```
nios2-linux-uclibc-gcc -v
```

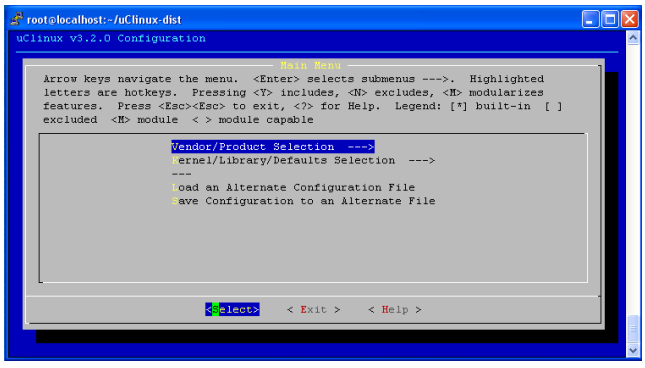
This should display

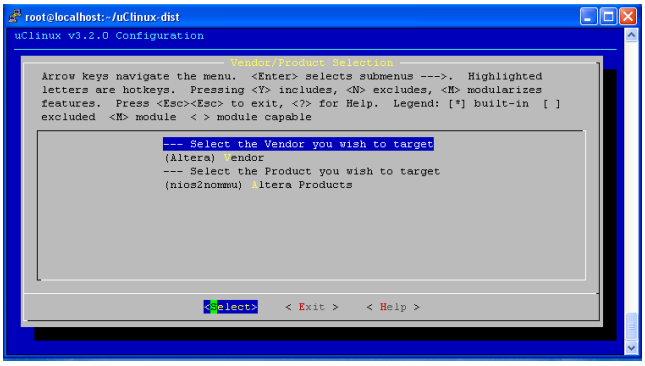
```
Reading specs from
   /opt/nios2/lib/gcc/nios2-linux-uclibc/3.4.6/specs
Configured with:
   /root/buildroot/toolchain_build_nios2/gcc-3.4.6/configure
--prefix=/opt/nios2 --build=i386-pc-linux-gnu
--host=i386-pc-linux-gnu --target=nios2-linux-uclibc
--enable-languages=c --enable-shared --disable-__cxa_atexit
--enable-target-optspace --with-gnu-ld --disable-nls
--enable-threads --disable-multilib --enable-cxx-flags=-static
Thread model: posix
gcc version 3.4.6
```

3.2 Building the uClinux Kernel

Once the toolchain is installed, the uClinux kernel can be built.

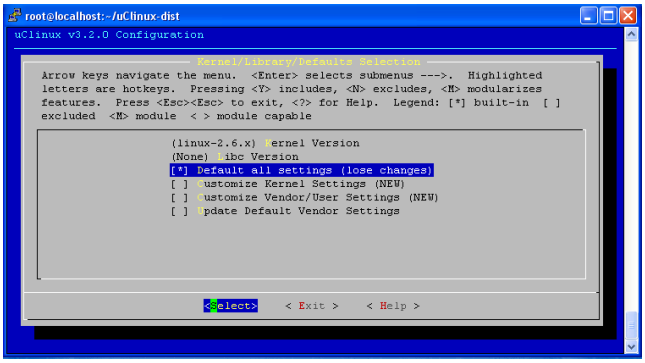
```
tar jxf uClinux-dist-20060803.tar.bz2
mv uClinux-dist-20060803-nios2-02.diff.gz uClinux-dist
cd uClinux-dist
gunzip -c uClinux-dist-20060803-nios2-02.diff.gz | patch -p0
make menuconfig
```





Verify you have the correct target.
Altera, nios2nommu, and Libc must be None.

```
Vendor/Product Selection --->
--- Select the Vendor you wish to target
(Altera) Vendor
--- Select the Product you wish to target
(nios2nommu) Altera ProductsĄă
```




```
Kernel/Library/Defaults Selection --->
(linux-2.6.x) Kernel Version
(None) Libc Version
[*] Default all settings (lose changes)
[ ] Customize Kernel Settings
[ ] Customize Vendor/User Settings
[ ] Update Default Vendor Settings
```

Then <exit> <exit> <yes>
Do not change any other setting until the first successful boot.
Next, set up memory and I/O addresses for the DE2 board.

```
make vendor_hwselect SYSPTF=~/toolchain/DE2_demonstrations/DE2_NET/nio
```

The *hwselect* script prepares the header files containing memory and I/O addresses for our board, the "nios2_system.h" header file. *hwselect* must be run before compiling the kernel. If these addresses are changed later, it is necessary to "make clean" in the kernel directory and run "make hwselect" again. Here is our project example that we use hwselect of DE2_NET/nios_0.ptf.

```
--- Please select which CPU you wish to build the kernel against:
(1) cpu_0 - Class: altera_nios2 Type: f Version: 6.0
Selection: 1

--- Please select a device to upload the kernel to:
(1) cfi_flash_0
Class: altera_avalon_cfi_flash
Size: 4194304 bytes
Selection: 1
```

```
--- Please select a device to execute kernel from:
(1) sram_0
    Class: sram_16bit_512k
    Size: 524288 bytes
(2) sdram_0
    Class: altera_avalon_new_sdram_controller
    Size: 8388608 bytes
(3) epcs_controller
    Class: altera_avalon_epcs_flash_controller
    Size: 2048 bytes
Selection: 2
```

This generates uClinux-dist/linux-2.6.x/include/nios2_system.h.

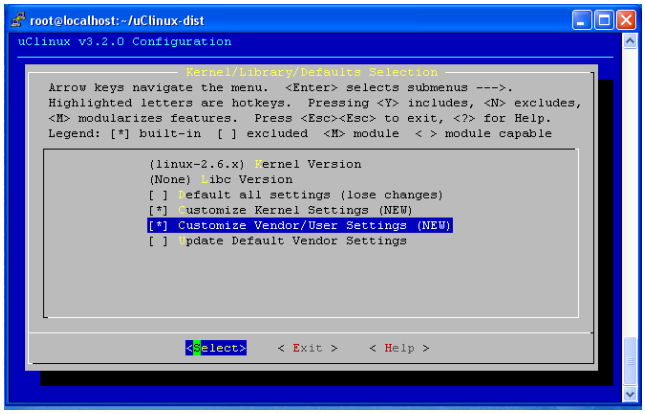
```
make romfs          # creates romfs dir
make       Ąă       # may fail; ignore
make                # should succeed
make linux image    # rebuild kernel for initramfs
```

This builds the compressed kernel in images/zImage in ELF format.

3.3 Customizing the Kernel and Adding System Applications

The kernel build procedure allows us to customize the kernel and select system applications such as telnetd. The following illustrates building the kernel with NIC functionality and *telnetd* and *ftpd* applications.

```
make menuconfig
```

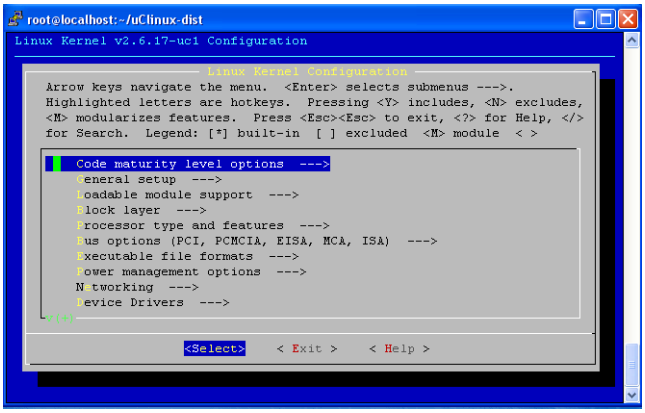



```
Kernel/Library/Defaults Selection --->
(linux-2.6.x) Kernel Version
(None) Libc Version
[ ] Default all settings (lose changes)
[*] Customize Kernel Settings
[*] Customize Vendor/User Settings
[ ] Update Default Vendor Settings
```

Then <exit> <exit> <yes>.

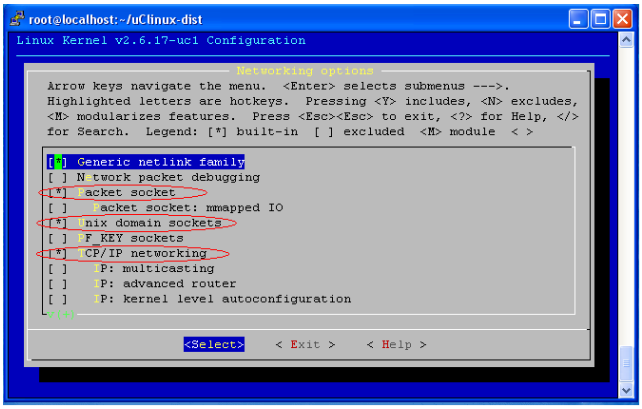
This will enter kernel configuration first, then enter user application configuration. You can select more applications in a second menu.

The first menu is kernel configuration.

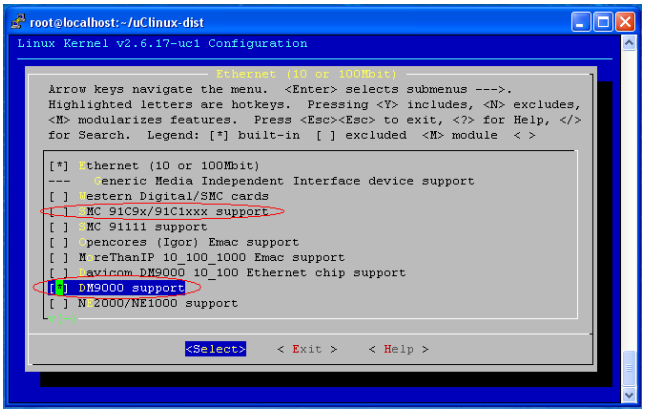


To enable network support,

```
Networking -->
[*] Networking support
Networking options --->
<*> Packet socket
<*> Unix domain sockets
[*] TCP/IP networking
```



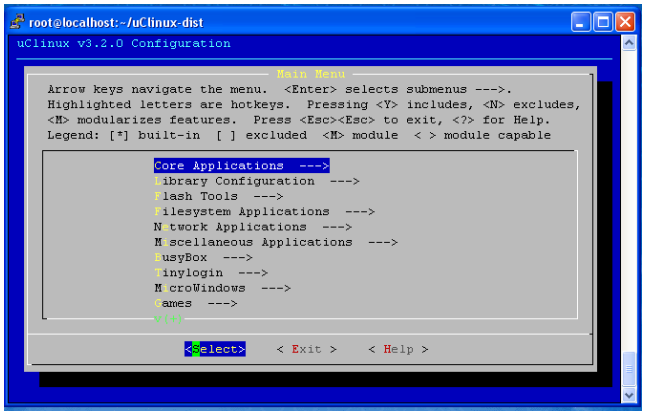

```
Device Drivers -->Network device support ŠĹą>
[*] Network device support
[*] Ethernet (10 or 100Mbit)
```




```
[*] Ethernet (10 or 100Mbit)
---   Generic Media Independent Interface device support
[ ] Western Digital/SMC cards
[*] SMC 91C9x/91C1xxx support      # For Altera NIOS dev board
[ ] Opencores (Igor) Emac support
[ ] MoreThanIP 10_100_1000 Emac support
[*] DM9000 support                 # For the DE2 board
```

Since we are using the DE2 board, we enable "DM9000 support," linux-2.6.x/drivers/net/dm9000.c. Rebuild the kernel and boot nios2 uclinux. It should detect the NIC device as eth0.

The second menu is application configuration. Here we add *ftpd* and *telnetd*.




```
Network Applications -->
[*] ftp
[ ] ftpd
[*] ftpd-new (0.17)
[*] telnetd
[*] telnetd does not use openpty()
```

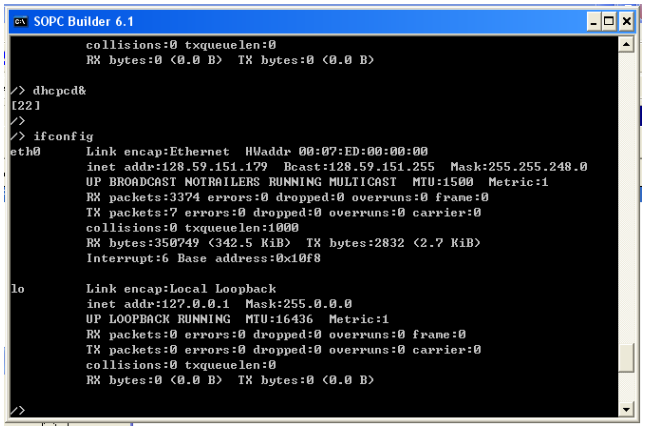
After downloading the image file, we open the Nios Command Shell located in `C:\altera\61\nios2eds\Nios II Command Shell.bat`. To load the image into DE2 FPGA board, we first run

```
nios2-download -g zImage
```

then run execute *nios-terminal*

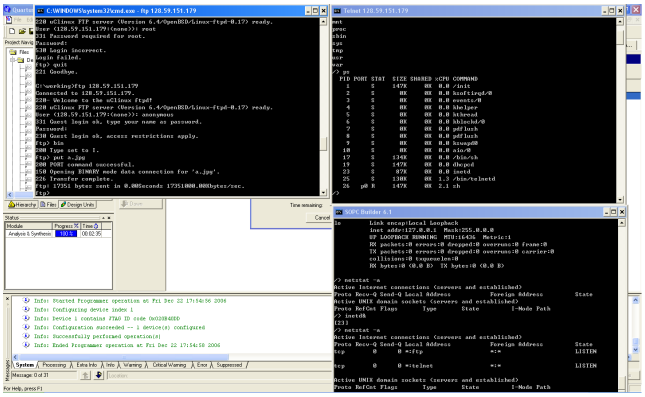
```
nios2-terminal
```

```
dhcpcd & # Obtain an IP address
```




```
inetd & # start inetd to invoke telnetd and ftpd services
```

The figure below shows both telnetd and ftpd working.



4 Writing User-space Programs

uClinux supports standard C libraries and system calls. Furthermore, hardware can be controlled directly from user-space programs by directly accessing memory-mapped I/O addresses. These are in the nios2_system.h header file.

There are two ways to download and run user space programs: use ftp to download the executable into the filesystem and run it from a command-line, or build it into romfs as part of the kernel. Here, we illustrate the second approach.

4.1 Hello World

To compile a simple program, add the -elf2flt flag. For example, create "hello.c"

```
#include <stdio.h>
int main(void)
{
  printf("hello world\\n");
}
```

and compile it with

```
nios2-linux-uclibc-gcc hello.c -o hello -elf2flt
```

The compiled object format is FLAT. You may check verify this with

```
nios2-linux-uclibc-flthdr hello
hello
    Magic:         bFLT
    Rev:           4
    Build Date:    Fri Dec 22 22:38:03 2006
    Entry:         0x40
    Data Start:    0x4a8c
    Data End:      0x5c48
    BSS End:       0x7ca8
    Stack Size:    0x1000
    Reloc Start:   0x5c48
    Reloc Count:   0x11e
    Flags:         0x1 ( Load-to-Ram )
```

Finally, copy hello to the rootfs's bin dir and rebuild the kernel image for initramfs.

```
cp hello ~/uClinux-dist/romfs/bin
cd ~/uClinux-dist-test
make linux image
```

Once uClinux is built, *hello* can now be run.

```
/bin> ./hello
hello world
/bin>
```

4.2 LED Access and Control

In this example, we access I/O locations for the green and red LEDs on the DE2, which has 8 green LEDs and 17 red ones. This user-space program takes two inputs: bits to control the green LEDs and bits for the red ones. We access the I/O locations through two constants defined in nios2_system.h: na_led_green and na_led_red.

```
#include <stdio.h>
#include "nios2_system.h"

short *green_led_address = na_led_green;
short *red_led_address = na_led_red;

int main(int argc, char *argv[]) {
  if (argc != 3) {
    printf("Arguement expected\n");
    return 0;
  }
  int ledGNum = atoi(argv[1]);
  int ledRNum = atoi(argv[2]);
  printf("Setting green led to %d, red led to %d\n",
         ledGNum, ledRNum);
  *green_led_address = (short) ledGNum;
  *red_led_address = (short) ledRNum;
}
```

4.3 Button interrupt by polling

We use polling to handle push button as well as interrupt register and show the result by turning on/off the led.

4.4 Multi-thread: Philosopher Dinning Programming

Here we use philosopher dinning programming to show how to write the multi-thread and how it works under uClinux.

5 Write System Level Program

We are going to discuss two system level programs, module programming and interrupt handler.

5.1 Module Programming

Before going to interrupt handler programming, we should discuss module programming first, because the interrupt handler is a kind of module programming.

5.2 Interrupt Handler

6 Conclusion and Future Work