A uClinux Primer

uClinux is the operating system of choice for many developers working with simple CPUs that do not feature a Memory Management Unit. Although closely related to its cousin, the Linux kernel, uClinux is different in subtle ways. Read more to know the whole story.

Why uClinux exists
uClinux is a derivative of the Linux kernel for MMU-less CPUs. To understand what an MMU is, and the facilities it offers, please see the section, “MMUs Demystified”. There are many CPUs, including DSPs that have a uClinux port, as of today. The Linux kernel, developed by hundreds of programmers around the world, is a stable UNIX like operating system featuring a mature TCP/IP stack, innumerous drivers for popular interconnect buses and devices. Although Linux itself has been ported to various CPU architectures like ARM, MIPS, SPARC, PowerPC, apart from x86, it does require an MMU to be present as part of the CPUs on which it runs. uClinux is a derivative that loses some features of the standard Linux kernel, while gaining the mature code base, development time and various features mentioned above.

MMUs Demystified
I will not be doing justice in compressing a full semester course into a few paragraphs, but if you know the basics of how programs are executed on any CPU, I'm sure you'll be able to understand this section with ease.

Modern operating systems depend a lot on the Memory Management Unit, an integral part of most CPUs. But what is an MMU, after all? Multitasking operating systems have to run many tasks simultaneously. But if you are talking strictly, it creates an illusion of running many tasks simultaneously. At any given point in time, there can be only one of two entities running on the CPU. It can either be various user tasks, or the operating system itself. To take any decision, or do any task(like fetch a packet of data from a network card), a piece of code, needs the CPU to execute it. In other words, unless a piece of code has the CPU, it can't do any task or supervise any activity in the system.

But now consider that a particular program you have written is running on the CPU currently. At exactly that point in time, does the operating system have the CPU? Not at all. It is your program that owns the CPU. At this same point of time, let us say you are trying to access a memory location that you are not supposed to access, how is the operating system going to know this? This is where the MMU steps in. These kinds of issues are extremely difficult to solve without assistance directly from hardware. When things like illegal memory access happen in systems that feature an MMU, the CPU stops whatever it has been doing so far and starts execution of an exception handler belonging to the operating system. Thus the operating system is immediately aware of things, as they go wrong and it then steps in to fix them.

What we just now discussed is something called protection. It is one of the reasons why the MMU exists. But it is by no means the only reason. There is another reason why it exists, and that is Virtual Memory. For protection to be effective, the operating system maintains tables in memory that describe for each task, the regions in memory allocated to it and access rights for each region, etc. For easy administration, memory is divided into pages that have a fixed size. A page size of 4k seems to be overwhelmingly popular, even across CPU architectures. Each time a process accesses memory, the CPU consults the information in the table to see if that process is allowed to access the memory it is trying to use. If there is a problem, as we saw, it executes operating system's exception handler.

To implement Virtual Memory(VM), programs are made to run with virtual addresses. These addresses are translated by the hardware, again using tables provided by the operating system, into physical addresses. For example, your program might be accessing 0x87664344, while the operating system would have mapped this to 0x66785564, which is the actual physical address. This scheme provides many advantages, out of which we will shortlist two. One very important thing is that this allows for non-contiguous memory pages to appear as contiguous to programs.This is achieved by mapping non-contiguous physical memory addresses into a contiguous virtual address space that a program will use. This allows the effects of Memory Fragmentation to diminish. The second advantage this arrangement offers, is that it allows least used pages of memory to reside else where, like say swap space on hard disk, when RAM availability is low. So, if a program tries to access a page that has been moved to swap space, it will cause an exception. The operating system's "page fault" exception handler then runs and tries to figure out the actual reason why the fault occurred. The operating system knows that the program tried to access an address that is valid, but is not present. It then brings back the page from swap space into memory(if necessary moving another page that it thinks is least used into swap space).

So, if you do not have an MMU, you can't have at least three things Linux programmers take for granted. There is no swap, so you need to remain within the bounds of available physical memory. You can't call malloc() and free() as you please. This may lead to fragmentation and your program might be unable to continue. You need to be careful with pointers, if you point to locations that are not valid in your program, you may be overwriting other programs, or the kernel address space!

fork(), the mother of all system calls

UNIX is notorious for its only way to create a process, the fork() system call. The fork() system call creates a new child process and runs the same program in it as the parent, and both the parent and child execute simultaneously. In uClinux however, there is no fork(), instead there is vfork(). The vfork() system call was invented due to the reason that fork() was a very expensive system call and that most processes simply call an exec() right after they have been forked. Now-a-days, Linux uses a technique called copy-on-write for forked processes. Most of the times, a newly forked process calls exec() to execute another program in the newly created process. So, there is no need to painstakingly copy the parent's address space as a whole. If the child writes to any of the variables, it is then that the kernel actually provides each one with a separate address space. This copying is an expensive process. So, for a newly forked process, the parent's address space is shared(mapped) with the child as read-only. This copy-on-write technique shines in situations where the child does an exec() as soon as it is forked. The copy-on-write technique depends on the capability of the MMU provide a read only mapping to the child as well as to detect writes to the common address space. In the days when such MMUs or the copy-on-write technique was unavailable, the vfork() system call was used to create child processes that would immediately call exec(). Thus the vfork() was the fork() system call's less expensive cousin.

The vfork() system call is different than fork in two crucial ways. Firstly, the parent's address space is shared with the child and secondly the parent's execution is blocked until the child does either of these two:

1. Calls exec()
2. Calls _exit()

After the child process does either of these two, the parent is scheduled to run again. The vfork() system call is not specific to uClinux. It is implemented in Linux as well. To demonstrate, let us look at this, simple program, vfork.c in listing 1. Type this out in your favorite editor, compile it and run it on your Linux machine.

--------------------- listing 1: vfork.c --------------
```
#include <stdio.h>
#include <unistd.h>

int main()
{
  int i;
  int pid;

  i=50;
  pid = vfork();
  if(pid==0){
    i=100;
    printf("Child here.\n");
    sleep(1);
    _exit(0);
  }
  else{
    printf("Parent here.\n");
    printf("Variable 'i' is %d.\n", i);
  }
}
```

This program demonstrates the two things mentioned earlier. One is that the parent blocks until the child either calls _exit() or exec(). In this example the child calls _exit(). To clearly notice that the parent does not run until the child terminates, there is a sleep() in the child. You can see that there is no activity from the parent for 1 second as well. The next thing discussed was that the address space is shared. The parent sets the value of the variable 'i' to 50, but the child changes that to 100. And after the child exits, the parent prints the value of 'i'. And it is the same value that the child sets. So, this illustrates that the address space is actually shared. You can try changing the vfork() to fork() and see what happens, to understand the differences more clearly.

Delegation of work to child processes
It is common practice in UNIX/Linux programming to create a child process and to delegate some work to it. The child process then does the work and terminates, returning the exit status to its parent. The fork() system call arranges the PID returned from the fork() system call to be zero in the child. Using only this condition, there is code in the program to execute in the child process. Since there is no fork() system call in uClinux, we need to live with vfork(). As discussed, earlier, vfork() causes the parent to block until the child either calls exec() or terminates. So, multitasking in the fork() system call sense is not

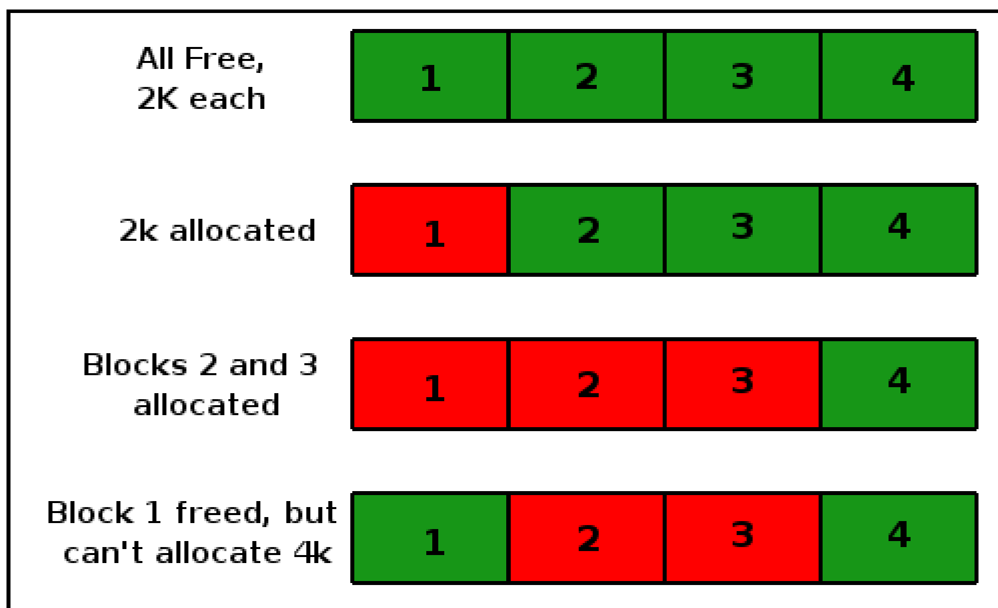
possible with vfork(). However, it is easy to circumvent this limitation. All that the child has to do, is to call the same program again with a special command line argument, for example "-child". In main, you need to check for this argument and take the appropriate action. This allows the parent to continue execution, since the child has actually called exec().

On memory and stack.
On Linux systems, there is a system call that programmers never use directly, but is actually used by the malloc() and free() library functions. The brk() system call is used to adjust the program's address space. In other words, it is used to shrink or grow it. There is one more library function, sbrk() that is used to increment the program's address space by the specified number of bytes. This actually depends on brk() in turn. When you got an MMU, there is a possibility to map or unmap pages of memory into a process address space, in uClinux, this is not possible. The stack and heap are quite different in uClinux. The stack is typically hardcoded to 4k. It is however possible to use compiler options and Flat format executable utilities to set the stack size in the program binary itself. The Flat format is defined by uClinux as a new executable file format and contains extra information needed for uClinux, such as the stack size of the application. For example, to set the stack size of a.out to 8k, do the following:

```
$ flthdr -s 8096 a.out
```

The other way is to define an environment variable from your makefile.

```
FLTFLAGS=-s 8096
```



[IMAGE frag.png Fig.1]
Memory fragmentation in MMU-less systems

uClinux systems also use a global pool of memory for use with applications. There is no per-process heap as with regular Linux systems. This leads to many problems, the most serious of which is fragmentation. Consider Fig.1. Let us consider that there is 8k of total available memory and that you allocate 2k. For easy undestanding, I have divided the memory logically into 2k blocks. You later allocate 4k. You later free up the 2k you allocated first, i.e block no. 1. Your total available memory now stands at 4k, but if you

were to now allocate 4k, the allocation would fail, since malloc() can't return a contiguous 4k block! This is a pitfall you need to be aware of. If there was an MMU, it would present the physically separate blocks in a contiguous virtual address space.

Trying out uClinux
What fun lies in all this theory if we do not do something real with it. So, let us try uClinux out! And since I didn't expect everyone to have hardware capable of running uClinux, We'll use a emulator to run uClinux. We will also see how to run the vfork.c example program on the emulator, by including it in the ROM image.

The uClinux port that we will be trying out is the Coldfire port. The Coldfire chips are derivatives of the famous Motorola 68000 family. There is an emulator by the same name for the Coldfire CPUs that is capable of running uClinux. I assume that your home directory is "/home/user" and that you have created a directory "/home/user/uclinux" as a place to put all downloaded files.

Getting uClinux
You can get the uClinux tar ball from:
http://www.uclinux.org/ports/coldfire/source.html
I got the 20060803 release from there.

Getting the m68k toolchain
Next you need to get the m68k toolchain from:
http://www.uclinux.org/pub/uClinux/m68k-elf-tools/
I got the m68k-elf-20030314 release, which is considered stable, you need to scroll down a bit to see where this is.

Getting the Coldfire emulator
You can then get the Coldfire emulator from:
http://www.slicer.ca/coldfire/index.php
Follow the "Download" link and get the latest version. That would be 0.3.1.

Setting things up

Untaring uClinux

$ tar xzf uClinux-dist-20060803.tar.gz

This will create a directory uClinux-dist

Installing the Tool Chain:

$ chmod +x m68k-elf-tools-20030314.sh
$ sudo ./m68k-elf-tools-20030314.sh

This will install the executables at /usr/local/bin

Setting up Coldfire:
$ tar xvzf coldfire-0.3.1.tar.gz
$ cd coldfire-0.3.1
$ ./configure
$ make
$ sudo make install

Compiling the sample application

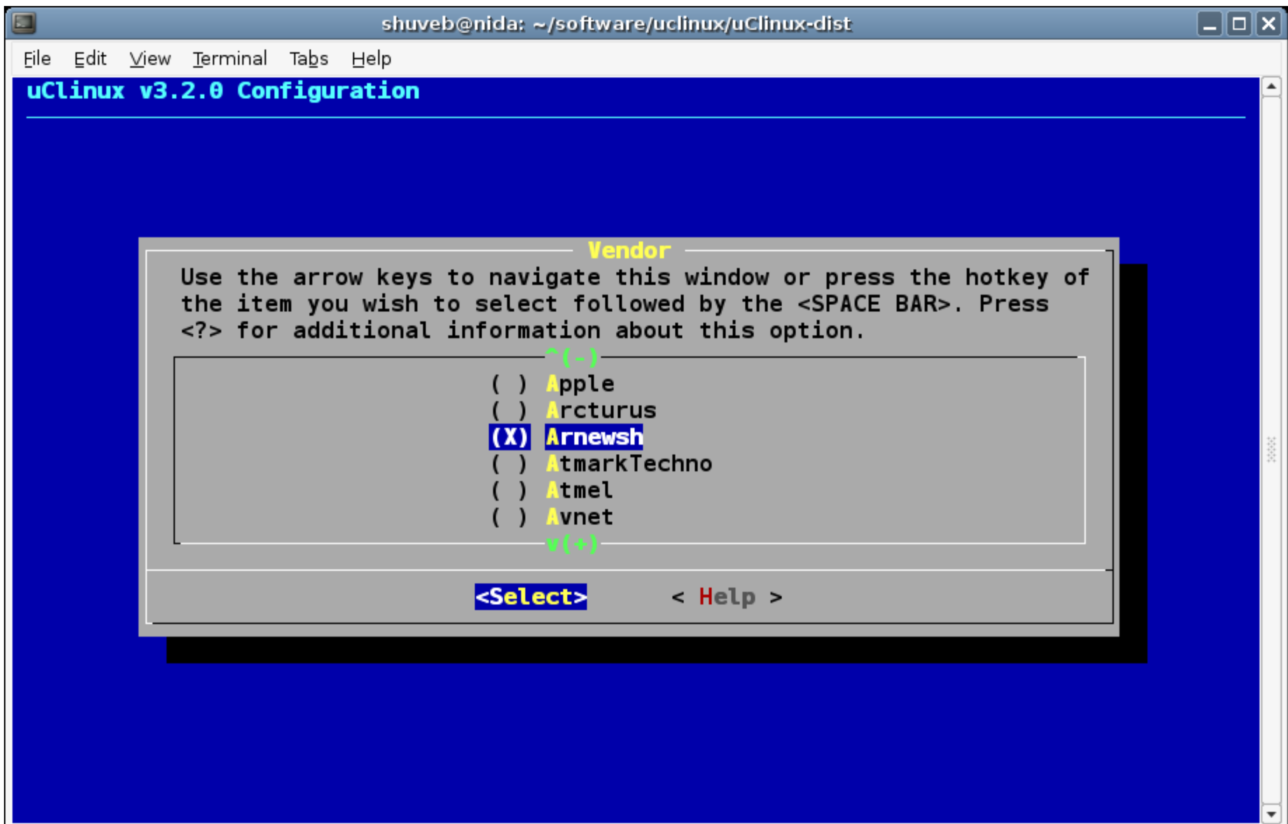
Let us now compile vfork.c presented in this article to try out on the emulator. Compile it with this command:

$ m68k-elf-gcc -m5200 -msep-data -Wl,-elf2flt -o vfork vfork.c -lc

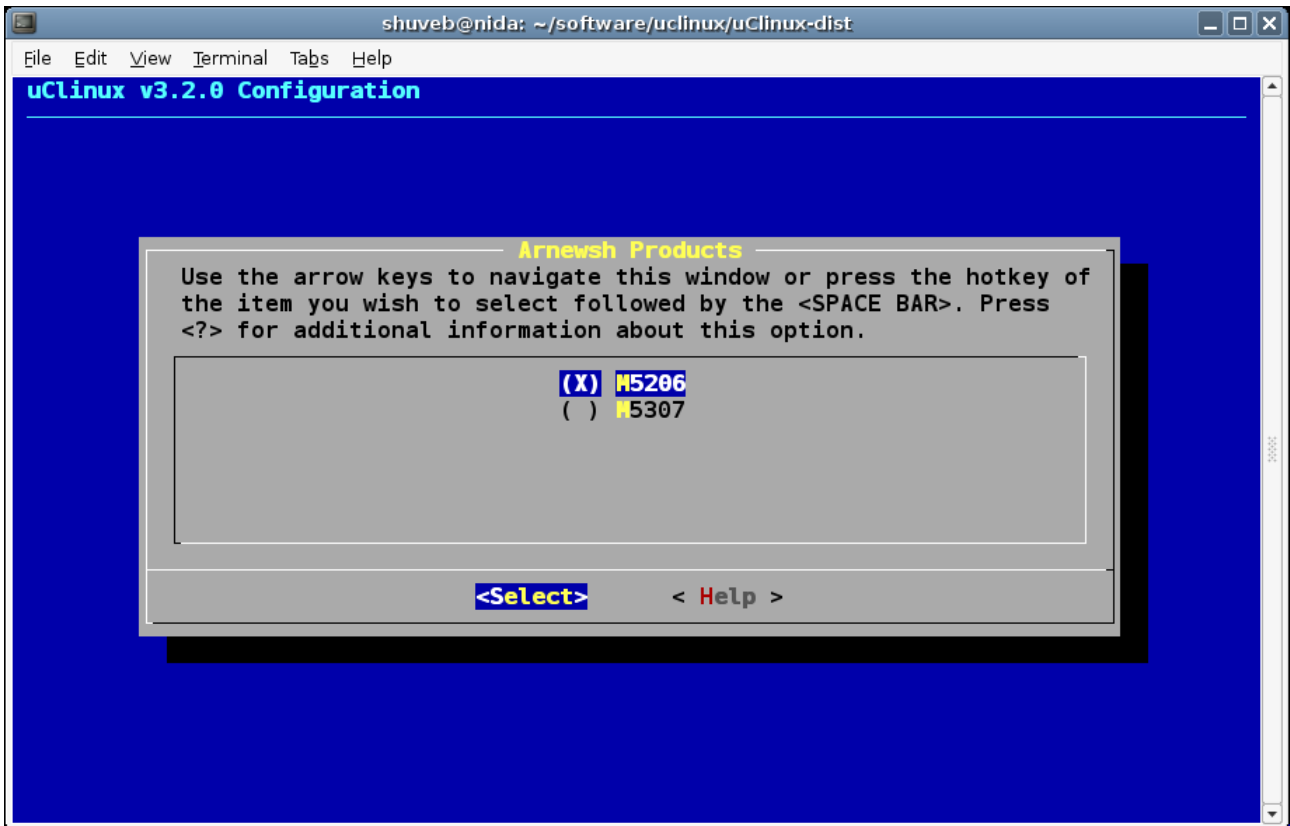
Setting up uClinux

$ cd /home/user/uclinux/uClinux-dist
$ make menuconfig

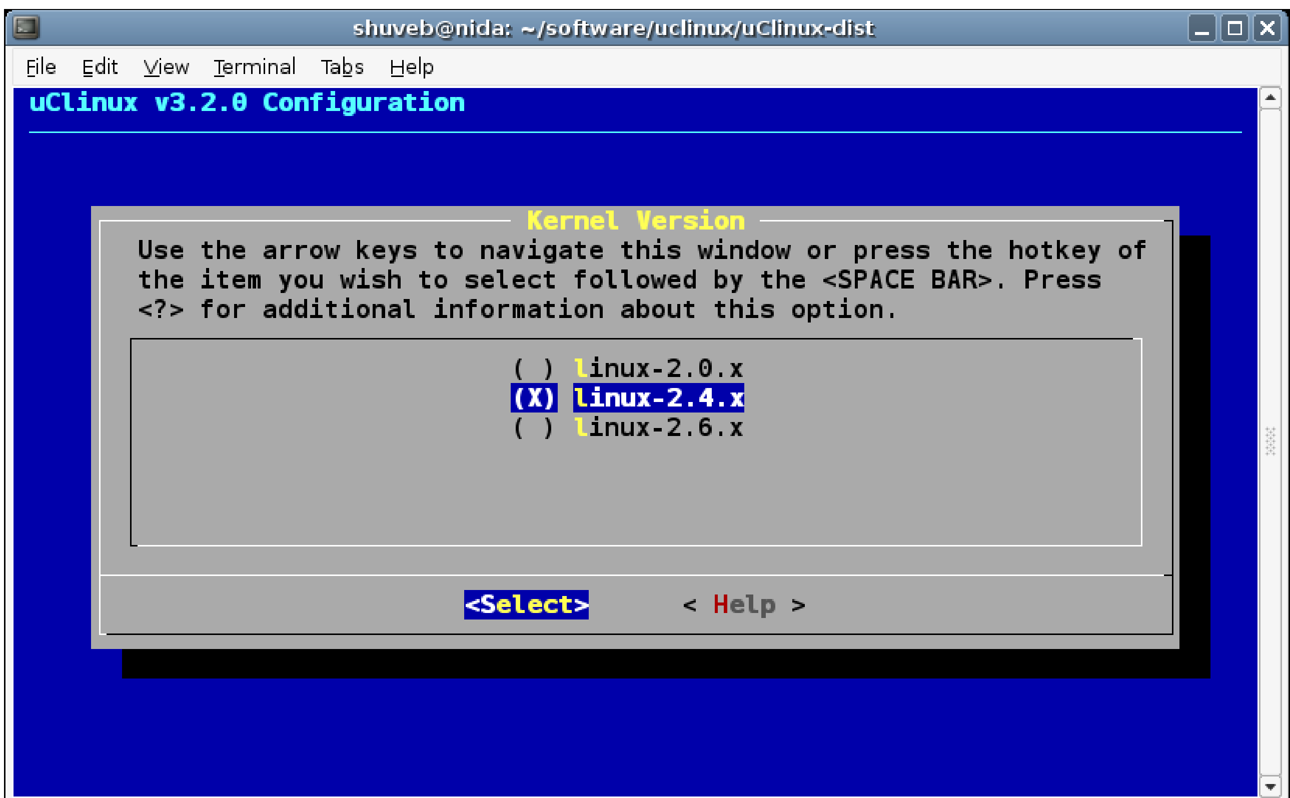
You will now be presented with a menu. You need to make just 3 settings to get started.



[IMAGE vendor.png]
Vendor/Product Selection -> Vendor -> Arnewsh



[IMAGE product.png]
Vendor/Product Selection -> Product -> M5206



[IMAGE kernel-ver.png]
Kernel/Library/Defaults Selection -> Kernel Version -> linux-2.4.x

```
$ make dep
$ make
```

This creates a kernel, a romfs image and a combined kernel + romfs image for use with boards. You might have had a look at the directory structure inside of uClinux-dist. Source code is organized as several directories. Some important ones are as follows:

Documentation	: uClinux notes and documentation
include	: include files
lib	: user space libraries
linux-2.0.x	: Linux 2.0 kernel series
linux-2.4.x	: Linux 2.4 kernel series
linux-2.6.x	: Linux 2.6 kernel series
uClibc	: uClibc, a simple C runtime library for small systems
user	: various user level applications including shells and utilities
vendors	: Vendor specific configuration files

Apart from these directories, there are two other directories created after compilation. These are the "images" and "romfs" directories. The "images" directory contains the kernel image, the romfs image and also a combined "image.bin" image after compilation completes successfully.

Running the image on the emulator
We can now use the newly compiled image and run it in the Coldfire emulator. Follow these steps:

```
$ cd /home/user/uclinux/uClinux-dist/images/
$ coldfire
[Now, fire up another terminal and do "telnet localhost 5206"]
Use CTRL-C (SIGINT) to cause autovector interrupt 7 (return to monitor)
Loading memory modules...
Loading board configuration...
Board ID: Arnewsh
CPU: 5206 (Motorola Coldfire 5206)
        unimplemented instructions: CPUSHL PULSE WDDATA WDEBUG
        62 instructions registered
        building instruction cache... done.
Memory segments: dram  timer1  timer2  uart1(on port 5206)
                 uart2(on port 5207)  sim  ne2000  isa
                 rom  sram
!!! Remember to telnet to the above ports if you want to see any output!
Hard Reset...
Initializing monitor...
Enter 'help' for help.
dBug> dl image.bin
Downloading Binary image... (offset=0x00010000)
dBug> go 10000
```

Now, in the terminal running telnet you will be able to see the new kernel you compiled boot up and drop you into a shell. You can press Ctrl+C in the terminal running Coldfire and press 0 to opt out of the emulator.

```
shuveb@nida: ~/software/uclinux/uClinux-dist/images
File  Edit  View  Terminal  Tabs  Help
Last modified Nov 1, 2000 by Paul Gortmaker
NE*000 ethercard probe at 0x40000300: not found (no reset ack).
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
CSLIP: code copyright 1989 Regents of the University of California.
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 2 disk images:
0: EC07C-12A47B [VIRTUAL EC07C-12A47B] (RO) <ROMFS>
1: FFE20000-FFE3FFFF [VIRTUAL FFE20000-FFE3FFFF] (RW) <NONE>
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
PPP generic driver version 2.4.2
PPP MPPE compression module registered
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 512 bind 512)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (romfs filesystem) readonly.
Freeing unused kernel memory: 24k freed (0xd2000 - 0xd7000)

Sash command shell (version 1.1.1)
/> ls -l
drwxr-xr-x  1 0        0              32  Jan  1 1970  bin
drwxr-xr-x  1 0        0              32  Jan  1 1970  dev
drwxr-xr-x  1 0        0              32  Jan  1 1970  etc
drwxr-xr-x  1 0        0              32  Jan  1 1970  home
drwxr-xr-x  1 0        0              32  Jan  1 1970  lib
drwxr-xr-x  1 0        0              32  Jan  1 1970  mnt
drwxr-xr-x  1 0        0              32  Jan  1 1970  proc
lrwxrwxrwx  1 0        0               8  Jan  1 1970  tmp -> /var/tmp
drwxr-xr-x  1 0        0              32  Jan  1 1970  usr
drwxr-xr-x  1 0        0              32  Jan  1 1970  var
/>
```

[IMAGE coldfire.png]

Including the example program in the ROM image

Follow these steps to include the example program, or any program you create into the ROM image so that you can try it out on the emulator.

$ cp /home/user/uclinux/vfork /home/user/uclinux/uClinux-dist/romfs/bin/
$ cd /home/user/uclinux/uClinux-dist/
$ make image

This will create image.bin in the "images" directory that will include your new vfork example program. You can, in this way write example programs and run them in the emulator running uClinux. You can boot up the Coldfire emulator using the new image and try out the vfork example by typing "vfork" in the shell.

I hope this article will motivate you enough to check out uClinux and also see the differences between the Linux kernel and uClinux.

Bio:

Shuveb Hussain loves the freedom offered by GNU/Linux. He works in the Embedded Linux Team for Novatium Solutions, Chennai. He blogs at http://binarykarma.org and can be reached through shuveb@gmail.com. This article first appeared in Linux For You magazine, India. http://www.lfymag.com