uClinux
Micro-Controller Linux

Greg Ungerer
gerg@snapgear.com
gerg@uclinux.org

SnapGear – A CyberGuard Company
825 Stanley St, Woolloongabba
QLD. 4102. Australia

www.snapgear.com

PH: +61 7 3435 2888
FAX: +61 7 3891 3630

Abstract

Micro-Controller Linux (uClinux) is an open source project that adds support to Linux that enable it to run on microprocessors without Memory Management Units (MMU). These types of processors have traditionally made up the bulk of processors used in embedded systems.

This paper will cover the basic architecture of uClinux, and in particular the design and code changes required to deal with not having any memory management. The kernel, device driver, library and application level changes will be detailed and explained. Many people will be surprised at how similar Linux is without an MMU!

What uClinux can do today will be covered, and this includes what hardware platforms are supported, what peripherals and what application and library packages have been ported. It will also cover what tools are required, and how to get setup for and develop with uClinux.

1. **Introduction**

> *Pronounced "you-see-linux", the name uClinux comes from combining the greek letter "mu" and the english capital "C". "Mu" stands for "micro", and the "C" is for "controller".*

Micro-controller Linux (uClinux for short) is an open source project to port Linux to microprocessors that do not have Memory Management Units (MMU). The goal is to create complete working systems, so this involves kernel, library and application level work.

As a project uClinux started in 1998 when Jeff Dione and Kenneth Albanowsky attempted a port of the Linux kernel to the Motorola 68328 Dragonball processor (this is the one used in the classic Palm Pilot PDA). It was using a 2.0.33 kernel. From here Greg Ungerer ported it to the Motorola ColdFire processor family at the end of 1998 and start of 1999. Soon after followed ports to ARM cores (I think the first was done by WireSpeed, maybe Joe DeBlequeue) and to the AXIS ETRAX architecture. Many other ports or architecture, boards and kernel versions have followed since.

The most extensive code changes are required to the Linux kernel. uClinux systems are true Linux systems though, the kernel support for running without an MMU is in an add on to Linux, it is not a different code base. We start with a stock Linux kernel source tree and add support for running without an MMU. So the uClinux kernel support is no more than a patch against standard Linux kernel sources.

Although the micro-controller market contains everything from 4bit to 64bit CPU architectures uClinux is targeted at the classic 32bit (and even 64bit) microprocessors. There is no support for 16bit or less CPU's.

The differentiation between a micro-controller and a standard CPU is blurry. A simplistic definition is any CPU that may be used in an embedded system could be considered a micro-controller. Better is any CPU that integrates a number of system peripherals with the CPU core is a micro-controller. Historically these types of CPUs are low cost or specialized for certain types of functions, and thus are not as full featured as their real computer counterparts. Often that meant they did not have features like memory management units. In recent years though the trend is to include MMUs, even on ultra low cost specialized CPU's. In any case uClinux is all about supporting CPUs that do not have MMUs.

Interestingly because uClinux is a set of additional patches for standard Linux sources all the existing CPU support for processors with MMUs is still present. The one kernel sources tree supports both processors with and without MMUs.

2. Kernel

The uClinux kernel is just the Linux kernel with support added for processors without MMUs. So, for the most part, you get the full Linux kernel feature set when you use uClinux. The Linux kernel API (in this case the system call set) is unchanged from standard Linux. Architecture implementation differences still apply, but in the same way as for all ports of Linux to non x86 architectures.

Currently there are two stable streams of uClinux kernel development and one experimental. Stable uClinux kernels exist based on 2.0.39 and 2.4.22, and the current cutting edge kernel is based on 2.6.0-test11 (at least as of this writing – it is probably newer by the time you read this).

The uClinux system is fully multi-tasking, with the usual process and process control model. All filesystems and related operations are identical. As is all networking support, and even the device driver interfaces are unchanged for uClinux. uClinux even supports dynamic kernel loadable modules.

Obviously some changes are required to the memory management sub-system of the kernel. Outside of architecture support this is the bulk of the uClinux patch. There is no notion of virtual memory (VM), and no form of memory protection between processes, between the kernel and processes or hardware device register sets. That is a fact of life without an MMU.

Not withstanding the different memory subsystem uClinux maintains the classic separation of user and kernel space. Each has its own stack, just as on a VM system, and if the hardware supports it the kernel maintains different privilege levels for each (although clearly it doesn't mean much when there is no memory protection!). Where hardware does not support privilege levels, or different mode stack pointers these are emulated in software.

A common question is whether uClinux needs less memory than a VM Linux system. In general the answer is no. But most uClinux systems are small by design, keeping their setup to a minimum. Practical uClinux systems can be built in as small as 1MB of RAM.

2.1 Supported Architectures

The range of CPU architectures and specific CPUs that uClinux supports is truly amazing. At the very least the list is:

- Motorola 68k family (68x302, 68306, 68x328, 68332, 68360)

- Motorola ColdFire (5206, 5206e, 5249, 5272, 5282, 5307, 5407)
- ARM (silicon from Atmel, NetSilicon, Aplio, TI, Samsung, Conexant, and more)
- Intel i960
- Sparc LEON
- MIPS (Brecis, …)
- NEC v850 family
- Hitachi H8/300
- Xilinx Microblaze (FPGA processor)
- Altera NIOS
- AXIS ETRAX
- Analog Devices Blackfin

There is more in development. The ones that I know about include:

- Hitachi Super SH2
- Motorola MCORE
- OpenCORES OpenRISC (FPGA)

There is probably more, the uClinux community is very active!

2.2Kernel Internals

The key difference between standard Linux and uClinux is the lack of any form hardware assisted memory management, that is the system supports no form of virtual memory. That implies no on demand loading, and that applications must wholly fit in RAM (or at least RAM and flash/ROM if executing in place). No current uClinux systems support swapping to any form of secondary storage either.

The underlying memory allocation system of Linux is used “as is”. The management of free and used areas of memory can be identical, it does not matter that virtual page mappings exist on top of used memory or not. The only change in this area is to allow the Linux allocator to keep regions of larger sizes available for allocation. When a memory allocation is requested in uClinux the kernel allocator needs to find a single contiguous chunk of RAM big enough to satisfy the request. It is not possible to virtually map a set of pages together to construct a larger region, so uClinux needs these larger allocation regions to satisfy large requests.

For the most part the virtual mapping support code is just stubbed out for uClinux. Virtual and physical addresses are treated as identical. Most kernel data structures associated with virtual memory support are left intact, and the internal function interfaces left unchanged. The changes made within the 2.6 series kernels to support

uClinux are clean and reasonably small, and well demonstrate the low overall impact adding MMUless support has had on the Linux kernel.

There are some interesting side effects of not having virtual memory in other parts of the kernel. It is worth going over those here, the four key ones are:

1. no easy way to implement real *fork()*
2. no way to dynamically grow an applications stack
3. no way to dynamically grow a heap (effects *sbrk()* system call)
4. memory fragmentation problems

Fork is more of problem that it would first seem. A true fork creates a mirror image of the current processes memory space, and then each of the parent and child get to execute in their own memory space. What one does has no effect on the other. Problem is that we have no notion of a virtual address space, when applications are running in uClinux they are all sharing the same address space with each other (and the kernel, and usually peripheral devices as well). When pointers are created, when call return addresses are pushed onto a stack, these are all absolute addresses. You cannot just copy the process memory image to another location, all these absolute addresses will now be wrong – pointing back into the parent's memory region. There is also no way to "fix" these absolute addresses as you copy, you just cannot tell what is really a pointer and what is random data.

For efficiency sake in uClinux we use the *vfork()* system call in place of *fork()*. With *vfork()* both parent and child share the memory region of the process. The semantics are that the child process runs to either *exec()* or *exit()* completion, the parent sleeps until then and resumes normal scheduled execution after that. The child process must be extremely careful to leave the parent memory region in a consistent state. *vfork()* has been around for years, originating from BSD UNIX. The reasoning behind it was that most programs *fork()* then do an *exec()* very soon after, effectively tearing down the copy of the memory space that was just copied in the *fork()*.

Without virtual memory we have no page mapping and there is no way to set markers for when the application stack becomes full. In uClinux fixed size stacks are allocated for each process at *exec()* time. The stack size is stored as part of the binary program file header, so it can be set on a program by program basis to minimize wasted memory.

Also without page table mappings in place we cannot dynamically grow a process heap in the conventional way. There is no simple way to implement the convention *sbrk()* system call that grows the heap contiguously. It is strait forward to allocate more memory, just not easy to make it contiguous with the current heap allocation. It turns out this is relatively easy to work around in the library code. The trick is to use *mmap()* to allocate memory instead of *sbrk()*. Using *mmap()* means the kernel will keep track of

the application allocated memory regions (which can be anywhere in the system address space) . This is nice, when the process exists it is simple to walk the list of associated *mmap* regions and free them back to the kernel free memory pool.

Lastly memory fragmentation is generally more of a problem under uClinux. When the kernel or a process tries to allocate a chunk of memory it must be fulfilled with a single contiguous chunk, that means that a single region of the right size needs to be found – separate smaller pages cannot be virtually mapped together to form the desired region size.

3. Libraries

There is one good reason you would not use glibc in uClinux systems, it is rather large! It has been done, but in practice no one uses it.

The preferred library for use in uClinux systems is uClibc. It is a descendant of the original uClinux library uC-libc. It is a collection of lightweight, standards compliant, functions that give you about 95% coverage of the glibc function set, but is much smaller. As a general rule anything that compiles and works on glibc will compile and work on uClibc. uClibc can be used on both MMU and MMUless systems. uClibc can be used as a shared or static library, and offers many advanced features like threading.

Some uClinux supported architectures support shared libraries. Currently they are only generally supported on m68k/ColdFire based systems. There has been at least one implementation of shared library support for ARM based uClinux systems, but this has never been made available as GPL open source (as far as I am aware).

Fundamentally two changes need to be made to a C library to support uClinux. For one *vfork()* needs to be implemented, and secondly the *malloc()* family of functions needs to be changed to use *mmap()* as the system call to get and free memory. Generally these are simple to do.

Many other libraries have also been ported to uClinux, The list includes openssl, libpcap, zlib, libjpeg, libpng, and many others.

4. Applications

Applications are loaded and run the same way under uClinux as Linux. Applications are made up of the same fundamental parts in uClinux too, they each have a code portion (sometimes called the text segment) an initialized data section (often called the data segment), an un-initialized data section (called the bss) and a stack.

One notion that is supported on many uClinux target architectures is the ability to leave the code section of an application in fixed random access storage (say something like a flash or ROM memory) and execute the instructions from that memory space. This is called “execute in place” (XIP) and can provide great memory savings. For this to be possible the entire code section of the program must be stored in one contiguous chunk. Not many filesystems actually do this.

Of-course uClinux also supports the more typical notion of loading a programs code and data into RAM and executing it from there.

4.1 FLAT Files

uClinux uses a new application binary file format called the flat file. The reasoning for a new file format is two fold. Primarily we want to simply the loading and running process for an application. Secondly we want a very small and lightweight binary format (we want to be able to build really small footprint systems).

On a virtual memory systems applications are absolutely linked to load and run in there own virtual memory space. Addresses within the code and data are fixed in that virtual address space. Generally we don’t have fixed addresses in uClinux. An application may be loaded and run anywhere in RAM, or when XIP at some location in flash/ROM. We will not know in advance at what memory address the code will actually reside in.

There is basically two different methods used in uClinux to deal with the unknown address problem.

1. Relocation

   Relocation entries are stored in the flat binary. When the program is being loaded to run the kernel flat loader (binfmt_flat) patches the code and data with the relocations (it uses the addresses range allocated for this application as the relocation address). Obviously for this method an applications code must be loaded into RAM, it cannot be run XIP in flash/ROM.

2. Position Independent Code (PIC)

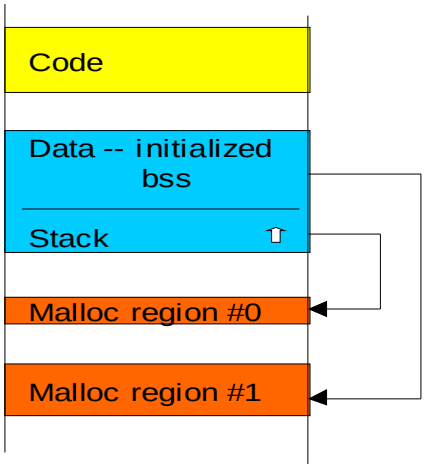
   When compiling the application we instruct the compiler to generate position independent code, that is code that has no absolute address references. It is not enough though to just have the code position independent though, we also need to have the data section position independent. This is typically achieved through the

use of a global offset table, where a table of address offsets is created for every address and all accesses are indexed through a base register.

PIC code often tends to be a little slower, due to the indirect access required. But for us it has the advantage of allowing sharing of code regions and for XIP. Note that every instance of a running application still has to have its own data segment and stack in RAM, only the code segment can be shared, or left and used in place in flash/ROM.

It would be fair to say that the PIC method is more popular in uClinux systems. But it cannot be supported on all architectures, and it does require a compiler capable of generating PIC code and data. Relocation is simpler to implement, and often is supported first on a new uClinux architecture port.

Relocation and PIC are not mutually exclusive, both can and often are supported on a system. The kernel loader can determine from the flat format file header whether the program can be run XIP or not.



This diagram is a simplistic representation of what a processes memory mapping might look like. Note that typically the data and stack regions are allocated as a single chunk, and notice that this is dislocated from the programs code section. The code section may well be in flash/ROM or some other polace in RAM – this is typical for XIP. It is also possible for a relocation load that the code, data and stack are allocated as a single chunk, and thus would be contiguous. Also note the malloced regions (that is what is conevntially referred to as the heap) is allocated from whatever free memory the kernel has available, again almost never contiguous to the processes code or data regions.

Although it would be possible to support ELF format applications on uClinux it has never been done. It would require relocating the code and data at load time – unlike on a

VM Linux kernel where it is already fully linked. So the Linux kernel ELF loader could not be used as it is.

The other advantage of flat format files is that they are extremely small. The header is 40 bytes, and no padding is used within the file.

4.2 Application Ports

The great thing about building a system on top of standard Linux and preserving the API is that you can port and use just about any application to uClinux that exists for Linux. The set of ported applications for uClinux is simply huge.

Here is a short list of ported application packages:

TOOLS/UTILITIES sash shell, minix shell, busybox, tinylogin, agetty, python, vi (clone), tip

NETWORKING net-tools, ping, ipfwadm/iptables, tftp, ftp, dhcpcd, traceroute, tcpdump, ssh, ntp, wget, iproute2

SERVERS init, inetd, pppd, pptpd, diald, boa (web), telnetd, tftpd, ftpd, dhcpd, samba, squid, snmpd, zebra, Freeswan (IPsec), dnsmasq, gdbserver, sshd

FILESYSTEM mount/umount (including NFS), smbmount/smbumount, e2fsprogs, fdisk, reiserfs tools,

MISCELLANEOUS mp3play, microwindows, mtd-utils, netflash, hotplug tools

This is but a sampling of the packages ported. The uClinux-dist distribution contains over 150 application packages currently that can run on uClinux.

5. Tools

Like any other Linux system uClinux systems are built using the standard GNU tools. Exact versions vary between architectures but currently many of the main-stream stable targets are using:

- binutils-2.14
- gcc-2.95.3
- gdb-5.0

On many targets the uClinux community has patched these tools to improve position independent code and data support, and support shared libraries. This is certainly true of the m68k and arm tool chains used for uClinux.

Moves are under way to update to more recent gcc versions (specifically 3.3) and to integrate many of the uClinux specific patches back into the gcc source base.

Gdb is interesting, for some architectures its simulator capabilities can be used to run uClinux. For example the ARMulator simulator extension of GDB can run uClinux in its own right. Makes a great development tool to get up to speed on uClinux on ARM platforms, or to develop without real hardware.

Gdb is also useful in other ways. Many of the embedded processors now days contain jtag, bdm or on-chip debug modules. Generally these can be driven by simple hardware dongles to parallel ports or similar on a PC. Many are supported through servers or with patches by gdb. Many offer advanced debug features like the ability to start and stop the CPU, set break points, dump and change memory. Many also allow programming flash memory in-circuit. All these features make debug a lot easier on these embedded platforms.

Gdb can also be used to debug uClinux applications. Normally this is done via a network debug arrangement, running the gdbserver stub on the uClinux target system.

Another of the key tools required for uClinux development is the *elf2flt* converter. *Elf2flt* converts a uClinux application that has been compiled as an ELF format object (as is normally done) to a uClinux flat format file. The conversion is actually reasonably strait forward

For those unfamiliar with developing for deeply embedded targets the usual setup is to cross compile for your target from a host development PC. This is true for uClinux, where the target system is almost never used as the development system. Most developers choose a Linux PC as their development system. It has been done on PowerPC based laptops as well. And for the truly disturbed you can even develop uClinux systems (compiling from source and all) under Windows using Cygwin.

6. Developing

Putting together a development environment to build uClinux systems for the first time can be quite a daunting task, even for seasoned developers. The best place to start is with the uClinux-dist source distribution package, and the pre-built binary tool chains on uclinux.org.

The uClinux-dist source package is an all-in-one source package. It includes uClinux kernels (currently 2.0.x, 2.4.x and 2.6.x kernels), the uC-libc and uClibc libraries, and a huge collection of ported application packages. A makefile setup spans all these components and allows you to build entire systems with this one large source tree.

The uClinux-dist package extends the familiar Linux kernel configuration framework and makes it simple to build for a supported platform. After installing the source and a tool chain a few simple clicks through the top level configuration and you can be building a uClinux image that is ready to run on your hardware.

The uClinux-dist framework also lets you drill down and configure the kernel options and to easily choose which applications to include in the final target image. The uClinux-dist really does make it simple to build complete systems, and saves you from having to build the individual system components separately (kernel, libraries, apps, etc).

Porting new applications to uClinux is often quite strait forward. Special attention is needed in dealing with the *fork()/vfork()* change, and some thought should be given to an initial stack size. Otherwise many programs can easily be cross-compiled and used on uClinux systems.

Relative to developing programs on normal desktop and server systems working with uClinux is somewhat more challenging. Most new developers to embedded systems find the disassociation of the development host and their target a little disconcerting.

7. Future Work

uClinux is a very active project, there is always a lot going in the uClinux community. The varying interests of developers in this space pulls development in a lot of directions.

Over the past year much effort has been put into getting the core uClinux support into the mainline 2.6 Linux kernel sources. This has worked out pretty well, with the core present, and the m68knommu, NEC v850 and H8/300 processor architectures also in. There is always much work to be done maintaining these in the mainline kernel sources.

More architecture ports are on going. Particularly interesting at the moment is the work going on porting uClinux to FPGA based processors. This is a really exciting area, and

progress here has been good. UClinux is currently running on the Xilinx Microblaze, Altera NIOS and OpenCORES OpenRISC soft cores. The Microblaze and NIOS ports are essentially complete, although not all NIOS code is yet present in the uClinux CVS.

New platforms with currently supported architectures are always being added to. This is generally pretty easy to do. Also peripheral support is ongoing. It is usually not difficult to port existing device drivers for Linux to new platforms. Almost always the most effort is required just to deal with architectural differences (endianess, address memory scheme, etc) than anything else.

Shared libraries are only freely supported on one architecture family current, m68k/ColdFire. It is a heavily request feature for other platforms. There needs to be some work done to get it supported on ARM platforms and others too.

Another feature that is often requested is Real Time support. The RTAI real time extensions for Linux have been ported to the at least one member of the ColdFire processor family running uClinux, and the RTlinux extensions where ported to an old version of uClinux for the m68k (Dragonball) based uCsimm.

It is often commented that one day all processors will just have MMUs. This may well be true. In the meantime we have uClinux to satisfy the need for advanced operating systems on typical deeply embedded systems. uClinux will need to be around for a long time, processors without MMUs are not going away any time soon.

References

http://www.uclinux.org
The central site for all things related to uClinux. A good place to get started. The direct link to the all-in-one uClinux sources is http://www.uclinux.org/pub/uClinux/dist/

http://cvs.uclinux.org
The main uClinux CVS repository. Always contains the latest and greatest uClinux kernel source code.

http://www.uclibc.org
The home of uClibc, lightweight C library used on most uClinux systems.

http://www.gnu.org
Home of the GNU project. Main repository for tool chain sources (binutils, gcc, gdb, etc)

http://www.ucdot.org
Good informational and news site for developers of embedded Linux, with a strong slant towards uClinux.