More Linux for Less

uClinux™ on a $5.00 (US) Processor

Michael Hennerich
Analog Devices
hennerich@blackfin.uclinux.org

Robin Getz
Analog Devices
rgetz@blackfin.uclinux.org

Abstract

While many in the Linux community focus on enterprise and multi-processor servers, there are also many who are working and deploying Linux on the network edge. Due to its open nature, and the ability to swiftly develop complex applications, Linux is rapidly becoming the number one embedded operating system. However, there are many differences between running Linux on a Quad processor system with 16Gig of Memory and 250Gig of RAID storage than a on a system where the total cost of hardware is less than the price of a typical meal.

1 Introduction

In the past few years, Linux™ has become an increasingly popular operating system choice not only in the PC and Server market, also in the development of embedded devices—particularly consumer products, telecommunications routers and switches, Internet appliances, and industrial and automotive applications.

The advantage of Embedded Linux is that it is a royalty-free, open source, compact solution that provides a strong foundation for an ever-growing base of applications to run on. Linux is a fully functional operating system (OS) with support for a variety of network and file handling protocols, a very important requirement in embedded systems because of the need to "connect and compute anywhere at anytime." Modular in nature, Linux is easy to slim down by removing utility programs, tools, and other system services that are not needed in the targeted embedded environment. The advantages for companies using Linux in embedded markets are faster time to market, flexibility, and reliability.

This paper attempts to answer several questions that all embedded developers ask:

- Why use a kernel at all?
- What advantages does Linux provide over other operating systems?
- What is the difference between Linux on x86 and low cost processors?
- Where can I get a kernel and how do I get started?
- Is Linux capable of providing real-time functionality?
- What are the possibilities to port a existing real-time application to a system running also Linux?

2 Why use a kernel at all

All applications require control code as support for the algorithms that are often thought of as the "real" program. The algorithms require data to be moved to and/or from peripherals, and many algorithms consist of more than one functional block. For some systems, this control code may be as simple as a "super loop" blindly processing data that arrives at a constant rate. However, as processors become more powerful, considerably more sophisticated control or signal processing may be needed to realize the processor's potential, to allow the processor to absorb the required functionality of previously supported chips, and to allow a single processor to do the work of many. The following sections provide an overview of some of the benefits of using a kernel on a processor.

2.1 Rapid Application Development

The use of the Linux kernel allows rapid development of applications compared to creating all of the control code required by hand. An application or algorithm can be created and debugged on an x86 PC using powerful desktop debugging tools, and using standard programming interfaces to device drivers. Moving this code base to an embedded linux kernel running on a low-cost embedded processor is trivial because the device driver model is exactly the same. Opening an audio device on the x86 Desktop is done in exactly the same was as on an embedded Linux system. This allows you to concentrate on the algorithms and the desired control flow rather than on the implementation details. Embedded Linux kernels and applications supports the use of C, C++, and assembly language, encouraging the development of code that is highly readable and maintainable, yet retaining the option of hand-optimizing if necessary.

2.2 Debugged Control Structures

Debugging a traditional hand-coded application can be laborious because development tools (compiler, assembler, and linker among others) are not aware of the architecture of the target application and the flow of control that results. Debugging complex applications is much easier when instantaneous snapshots of the system state and statistical runtime data are clearly presented by the tools. To help offset the difficulties in debugging software, embedded Linux kernels are tested with the same tests that many desktop distributions use before releasing a Linux kernel. This ensure that the embedded kernel is as bug-free as possible.

2.3 Code Reuse

Many programmers begin a new project by writing the infrastructure portions that transfers data to, from, and between algorithms. This necessary control logic usually is created from scratch by each design team and infrequently reused on subsequent projects. The Linux kernel provides much of this functionality in a standard, portable, and reusable manner. Furthermore, the kernel and its tight integration with the GNU development and debug tools are designed to promote good coding practice and organization by partitioning large applications into maintainable and comprehensible blocks. By isolating the functionality of subsystems, the kernel helps to prevent the morass all too commonly found in systems programming. The kernel is designed specifically to take advantage of commonality in user applications and to encourage code reuse. Each thread of execution is created from a user-defined template, either at boot time or dynamically by another thread. Multiple threads can be created from the same template, but the state associated with each created instance of the thread

remains unique. Each thread template represents a complete encapsulation of an algorithm that is unaware of other threads in the system unless it has a direct dependency.

2.4 Hardware Abstraction

In addition to a structured model for algorithms, the Linux kernel provides a hardware abstraction layer. Presented programming interfaces allow you to write most of the application in a platform-independent, high-level language (C or C++). The Linux Application Programming Interface (API) is identical for all processors which support Linux, allowing code to be easily ported to a different processor core. When porting an application to a new platform, programmers must only address the areas necessarily specific to a particular processor—normally device drivers. The Linux architecture identifies a crisp boundary around these subsystems and supports the traditionally difficult development with a clear programming framework and code generation. Common devices can use the same driver interface (for example a serial port driver may be specific for a certain hardware, but the application $\longleftrightarrow$ serial port driver interface should be exactly the same, providing a well-defined hardware abstraction, and making application development faster).

2.5 Partitioning an Application

A Linux application or thread is an encapsulation of an algorithm and its associated data. When beginning a new project, use this notion of an application or thread to leverage the kernel architecture and to reduce the complexity of your system. Since many algorithms may be thought of as being composed of subalgorithm building blocks, an application can be partitioned into smaller functional units that can be individually coded and tested. These building blocks then become reusable components in more robust and scalable systems.

You define the behavior of Linux applications by creating the application. Many application or threads of the same type can be created, but for each thread type, only one copy of the code is linked into the executable code. Each application or thread has its own private set of variables defined for the thread type, its own stack, and its own C run-time context.

When partitioning an application into threads, identify portions of your design in which a similar algorithm is applied to multiple sets of data. These are, in general, good candidates for thread types. When data is present in the system in sequential blocks, only one instance of the thread type is required. If the same operation is performed on separate sets of data simultaneously, multiple threads of the same type can coexist and be scheduled for prioritized execution (based on when the results are needed).

2.6 Scheduling

The Linux kernel can be a preemptive multitasking kernel. Each application or thread begins execution at its entry point. Then, it either runs to completion or performs its primary function repeatedly in an infinite loop. It is the role of the scheduler to preempt execution of a an application or thread and to resume its execution when appropriate. Each application or thread is given a priority to assist the scheduler in determining precedence.

The scheduler gives processor time to the thread with the highest priority that is in the ready state. A thread is in the ready state when it is not waiting for any system resources it has requested.

2.7 Priorities

Each application or thread is assigned a dynamically modifiable priority. An application is limited to forty (40) priority levels. However, the number of threads at each priority is limited, in practice, only by system memory. Priority level one is the highest priority, and priority thirty is the lowest. The system maintains an idle thread that is set to a priority lower than that of the lowest user thread.

Assigning priorities is one of the most difficult tasks of designing a real-time preemptive system. Although there has been research in the area of rigorous algorithms for assigning priorities based on deadlines (for example, rate-monotonic scheduling), most systems are designed by considering the interrupts and signals triggering the execution, while balancing the deadlines imposed by the system's input and output streams.

2.8 Preemption

A running thread continues execution unless it requests a system resource using a kernel system call. When a thread requests a signal (semaphore, event, device flag, or message) and the signal is available, the thread resumes execution. If the signal is not available, the thread is removed from the ready queue; the thread is blocked. The kernel does not perform a context switch as long as the running thread maintains the highest priority in the ready queue, even if the thread frees a resource and enables other threads to move to the ready queue at the same or lower priority. A thread can also be interrupted. When an interrupt occurs, the kernel yields to the hardware interrupt controller. When the ISR completes, the highest priority thread resumes execution.

2.9 Application and Hardware Interaction

Applications should have minimal knowledge of hardware; rather, they should use device drivers for hardware control. A application can control and interact with a device in a portable and hardware abstracted manner through a standard set of APIs.

The Linux Interrupt Service Routine framework encourages you to remove specific knowledge of hardware from the algorithms encapsulated in threads. Interrupts relay information to threads through signals to device drivers or directly to threads. Using signals to connect hardware to the algorithms allows the kernel to schedule threads based on asynchronous events. The Linux run-time environment can be thought of as a bridge between two domains, the thread domain and the interrupt domain. The interrupt domain services the hardware with minimal knowledge of the algorithms, and the thread domain is abstracted from the details of the hardware. Device drivers and signals bridge the two domains.

2.10 Downside of using a kernel

- Memory consumption: to have a usable Linux system, you should consider having at least 4–8 MB of SDRAM, and at least 2MB of Flash.
- Boot Time: the kernel is fast, but sometimes not fast enough, expect to have a 2–5 second boot time.
- Interrupt Latency: On occasions, a Linux device driver, or even the kernel, will disable interrupts. Some critical kernel operations can not be interrupted, and it is unfortunate, but interrupts must be turned off for a bit. Care has been taken to keep critical regions as short as possible as they

cause increased and variable interrupt latency.

- Robustness: although a kernel has gone through lots of testing, and many people are using it, it is always possible that there are some undiscovered issues. Only you can test it in the configuration that you will ship it.

3 Advantages of Linux

Despite the fact that Linux was not originally designed for use in embedded systems, it has found its way into many embedded devices. Since the release of kernel version 2.0.x and the appearance of commercial support for Linux on embedded processors, there has been an explosion of embedded devices that use Linux as their OS. Almost every day there seems to be a new device or gadget that uses Linux as its operating system, in most cases going completely unnoticed by the end users. Today a large number of the available broadband routers, firewalls, access points, and even some DVD players utilize Linux, for more examples see Linuxdevices.[1]

Linux offers a huge amount of drivers for all sorts of hardware and protocols. Combine that with the fact that Linux does not have run-time royalties, and it quickly becomes clear why there are so many developers using Linux for their devices. In fact, in a recent embedded survey, 75% of developers indicated they are using, or are planning on using an open source operating system.[2]

Many commercial and non-commercial Linux kernel trees and distributions enable a wide variety of choices for the embedded developer.

One of the special trees is the uClinux (Pronounced *you-see-linux*, the name uClinux comes from combining the greek letter *mu* (μ) and the English capital *C*. *Mu* stands for *micro*, and the *C* is for *controller*) kernel tree, at `http://www.uclinux.org`. This is a distribution which includes a Linux kernel optimized for low-cost processors, including processors without a Memory Management Unit (MMU). While the nommu kernel patch has been included in the official Linux 2.6.x kernel, the most up-to-date development activity and projects can be found at uClinux Project Page and Blackfin/uClinux Project Page[3]. Patches such as these are used by commercial Linux vendors in conjunction with their additional enhancements, development tools, and documentation to provide their customers an easy-to-use development environment for rapidly creating powerful applications on uClinux.

Contrary to most people's understanding, uClinux is not a "special" Linux kernel tree, but the name of a distribution, which goes through testing on low-cost embedded platforms.

`www.uclinux.org` provides developers with a Linux distribution that includes different kernels (2.0.x, 2.4.x, 2.6.x) along with required libraries; basic Linux shells and tools; and a wide range of additional programs such as web server, audio player, programming languages, and a graphical configuration tool. There are also programs specially designed with size and efficiency as their primary considerations. One example is busybox, a multicall binary, which is a program that includes the functionality of a lot of smaller programs and acts like any one of them if it is called by the appropriate name. If busybox is linked to `ls` and contains the `ls`

[1] `http://www.linuxdevices.org`

[2] Embedded systems survey `http://www.embedded.com/showArticle.jhtml?articleID=163700590`

[3] `http://www.blackfin.uclinux.org`

code, it acts like the `ls` command. The benefit of this is that busybox saves some overhead for unique binaries, and those small modules can share common code.

In general, the uClinux distribution is more than adequate enough to compile a full Linux image for a communication device, like a router, without writing a single line of code.

4 Differences between MMU Linux and noMMU Linux

Since Linux on processors with MMU and without MMU are similar to UNIX256 in that it is a multiuser, multitasking OS, the kernel has to take special precautions to assure the proper and safe operation of up to thousands of processes from different users on the same system at once. The UNIX security model, after which Linux is designed, protects every process in its own environment with its own private address space. Every process is also protected from processes being invoked by different users. Additionally, a Virtual Memory (VM) system has additional requirements that the Memory Management Unit (MMU) must handle, like dynamic allocation of memory and mapping of arbitrary memory regions into the private process memory.

Some processors, like Blackfin, do not provide a full-fledged MMU. These processors are more power efficient and significantly cheaper than the alternatives, while sometimes having higher performance.

Even on processors featuring Virtual Memory, some system developers target their application to run without the MMU turned on, because noMMU Linux can be significantly faster than Linux on the same processor. Overhead of MMU operations can be significant. Even when a MMU is available, it is sometimes not used in systems with high real-time constraints. Context switching and Inter Process Communication (IPC) can also be several times faster on uClinux. A benchmark on an ARM9 processor, done by H.S. Choi and H.C. Yun, has proven this.[4]

To support Linux on processors without an MMU, a few trade-offs have to be made:

1. No real memory protection (a faulty process can bring the complete system down)
2. No fork system call
3. Only simple memory allocation
4. Some other minor differences

4.1 Memory Protection

Memory protection is not a real problem for most embedded devices. Linux is a very stable platform, particularly in embedded devices, where software crashes are rarely observed. Even on a MMU-based system running Linux, software bugs in the kernel space can crash the whole system. Since Blackfin has memory protection, but not Virtual Memory, Blackfin/uClinux has better protection than other noMMU systems, and does provide some protection from applications writing into peripherals, and therefore will be more robust than uClinux running on different processors.

There are two most common principal reasons causing uClinux to crash:

- Stack overflow: When Linux is running on an architecture where a full MMU exists, the MMU provides Linux programs

[4] `http://opensrc.sec.samsung.com/document/uc-linux-04_sait.pdf`

basically unlimited stack and heap space. This is done by the virtualization of physical memory. However most embedded Linux systems will have a fixed amount of SDRAM, and no swap, so it is not really "unlimited." A program with a memory leak can still crash the entire system on embedded Linux with a MMU and virtual memory.

Because noMMU Linux can not support VM, it allocates stack space during compile time at the end of the data for the executable. If the stack grows too large on noMMU Linux, it will overwrite the static data and code areas. This means that the developer, who previously was oblivious to stack usage within the application, must now be aware of the stack requirements.

On gcc for Blackfin, there is a compiler option to enable stack checking. If the option `-fstack-limit-symbol=_stack_start` is set, the compiler will add in extra code which checks to ensure that the stack is not exceeded. This will ensure that random crashes due to stack corruption/overflow will not happen on Blackfin/uClinux. Once an application compiled with this option and exceeds its stack limit, it gracefully dies. The developer then can increase the stack size at compile time or with the `flthdr` utility program during runtime. On production systems, stack checking can either be removed (increase performance/reduce code size), or left in for the increase in robustness.

- Null pointer reference: The Blackfin MMU does provide partial memory protection, and can segment user space from kernel (supervisor) space. On Blackfin/uClinux, the first 4K of memory starting at NULL is reserved as a buffer for bad pointer dereferences. If an application uses a uninitialized pointer that reads or writes into the first 4K of memory, the application will halt. This will ensure that random crashes due to uninitialized pointers are less likely to happen. Other implementations of noMMU Linux will start writing over the kernel.

4.2 No Fork

The second point can be little more problematic. In software written for UNIX or Linux, developers sometimes use the fork system call when they want to do things in parallel. The `fork()` call makes an exact copy of the original process and executes it simultaneously. To do that efficiently, it uses the MMU to map the memory from the parent process to the child and copies only those memory parts to that child it writes. Therefore, uClinux cannot provide the `fork()` system call. It does, however, provide `vfork()`, a special version of `fork()`, in which the parent is halted while the child executes. Therefore, software that uses the `fork()` system call has to be modified to use either `vfork()` or POSIX threads that uClinux supports, because they share the same memory space, including the stack.

4.3 Memory Allocation

As for point number three, there usually is no problem with the malloc support noMMU Linux provides, but sometimes minor modifications may have to be made. Memory allocation on uClinux can be very fast, but on the other hand a process can allocate all available memory. Since memory can be only allocated in contiguous chunks, memory fragmentation can be sometimes an issue.

4.4 Minor Differences

Most of the software available for Linux or UNIX (a collection of software can be found on `http://freshmeat.net`) can be directly compiled on uClinux. For the rest, there is usually only some minor porting or tweaking to do. There are only very few applications that do not work on uClinux, with most of those being irrelevant for embedded applications.

5 Developing with uClinux

When selecting development hardware, developers should not only carefully make their selection with price and availability considerations in mind, but also look for readily available open source drivers and documentation, as well as development tools that makes life easier—e.g., kernel, driver and application debugger, profiler, strace.

/subsectionTesting uClinux Especially when developing with open source—where software is given as-is—developers making a platform decision should also carefully have a eye on the test methodology for kernel, drivers, libraries, and toolchain. After all, how can a developer, in a short time, determine if the Linux kernel running on processor A is better or worse than running on processor B?

The simplest way to test a new kernel on a new processor is to just boot the platform, and try out the software you normally run. This is an important test, because it tests most quickly the things that matter most, and you are most likely to notice things that are out of the ordinary from the normal way of working. However, this approach does not give widespread test coverage; each user tends to use the GNU/Linux system only for a very limited range of the available functions it offers and it can take significant time to build the processor tool chain, build the kernel, and download it to the target for the testing.

Another alternative is to run test suites. These are software packages written for the express purpose of testing, and they are written to cover a wide range of functions and often to expose things that are likely to go wrong.

The Linux Test Project (LTP), as an example, is a joint project started by SGI and maintained by IBM, that has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP test suite contains a collection of tools for testing the Linux kernel and related features. Analog Devices, Inc., sponsored the porting of LTP to architectures supported by noMMU Linux.

Testing with test suites applies not only the kernel, also all other tools involved during the development process. If you can not trust your compiler or debugger, then you are lost. Blackfin/uClinux uses DejaGnu to ease and automate the over 44,000 toolchain tests, and checking of their expected results while running on target hardware. In addition there are test suites included in Blackfin/uClinux to do automated stress tests on kernel and device drivers using expect scripts. All these tests can be easily reproduced because they are well documented.

Here are the test results for the Blackfin gcc-4.x compiler.

=== gas Summary ===	
# of expected passes	79
== binutils Summary ===	
# of expected passes	26
# of untested testcases	7
=== gdb Summary ===	
# of expected passes	9018
# of unexpected failures	62
# of expected failures	41
# of known failures	27
# of unresolved testcases	9
# of untested testcases	5
# of unsupported tests	32
=== gcc Summary ===	
# of expected passes	36735
# of unexpected failures	33
# of unexpected successes	1
# of expected failures	75
# of unresolved testcases	28
# of untested testcases	28
# of unsupported tests	393
=== g++ Summary ===	
# of expected passes	11792
# of unexpected failures	10
# of unexpected successes	1
# of expected failures	67
# of unresolved testcases	14
# of unsupported tests	165

All of the unexpected failures have been analysed to ensure that the toolchain is as stable as possible with all types of software that someone could use it with.

6 Where can I get uClinux and how do I get started?

Normally, the first selection that is made once Linux is chosen as the embedded operating system, is to identify the lowest cost processor that will meet the performance targets. Luckily, many silicon manufacturers are fighting for this position.

During this phase of development it is about the 5 processor Ps.

- Penguins
- Price
- Power
- Performance
- Peripherals

6.1 Low-cost Processors

Although the Linux kernel supports many architectures, including alpha, arm, frv, h8300, i386, ia64, m32r, m68k, mips, parisc, powerpc, s390, sh, sparc, um, v850, x86_64, and xtensa, many Linux developers are surprised to hear of a recent new Linux port to the Blackfin Processor.

Blackin Processors combine the ability for real-time signal processing and the functionality of microprocessors, fulfilling the requirements of digital audio and communication applications. The combination of a signal processing core with traditional processor architecture on a single chip avoids the restrictions, complexity, and higher costs of traditional heterogeneous multi-processor systems.

All Blackfin Processors combine a state-of-the-art signal processing engine with the advantages of a clean, orthogonal RISC-like microprocessor instruction set and Single Instruction Multiple Data (SIMD) multimedia capabilities into a single instruction set architecture. The Micro Signal Architecture (MSA) core is a

dual-MAC (Multiply Accumulator Unit) modified Harvard Architecture that has been designed to have unparalleled performance on typical signal processing[5] algorithms, as well as standard program flow and arbitrary bit manipulation operations mainly used by an OS.

The single-core Blackfin Processors have two large blocks of on-chip memory providing high bandwidth access to the core. These memory blocks are accessed at full processor core speed (up to 756MHz). The two memory blocks sitting next to the core, referred to as L1 memory, can be configured either as data or instruction SRAM or cache. When configured as cache, the speed of executing external code from SDRAM is nearly on par with running the code from internal memory. This feature is especially well suited for running the uClinux kernel, which doesn't fit into internal memory. Also, when programming in C, the memory access optimization can be left up to the core by using cache.

6.2 Development Environment

A typical uClinux development environment consists of a low-cost Blackfin STAMP board, and the GNU Compiler Collection (gcc cross compiler) and the binutils (linker, assembler, etc.) for the Blackfin Processor. Additionally, some GNU tools like `awk`, `sed`, `make`, `bash`, etc., plus `tcl/tk` are needed, although they usually come by default with the desktop Linux distribution.

An overview of some of the STAMP board features are given below:

- ADSP-BF537 Blackfin device with JTAG interface
- 500MHz core clock
- Up to 133MHz system clock
- 32M x 16bit external SDRAM (64MB)
- 2M x 16bit external flash (4MB)
- 10/100 Mbps Ethernet Interface (via on-chip MAC, connected via DMA)
- CAN Interface
- RS-232 UART interface with DB9 serial connector
- JTAG ICE 14 pin header
- Six general-purpose LEDs, four general-purpose push buttons
- Discrete IDC Expansion ports for all processor peripherals

All sources and tools (compiler, binutils, gnu debugger) needed to create a working uClinux kernel on the Blackfin Processors can be freely obtained from `http://www.blackfin.uclinux.org`. To use the binary rpms, a PC with a Linux distribution like Red Hat or SuSE is needed. Developers who can not install Linux on their PC have a alternative. Cooperative Linux (coLinux) is a relatively new means to provide Linux services on a Windows host. There already exists an out-of-the-box solution that can be downloaded for free from `http://blackfin.uclinux.org/projects/bfin-colinux`. This package comes with a complete Blackfin uClinux distribution, including all user-space applications and a graphical Windows-like installer.

After the installation of the development environment and the decompression of the uClinux distribution, development may start.

[5] Analog Devices, Inc. Blackfin Processors `http://www.analog.com/blackfin`

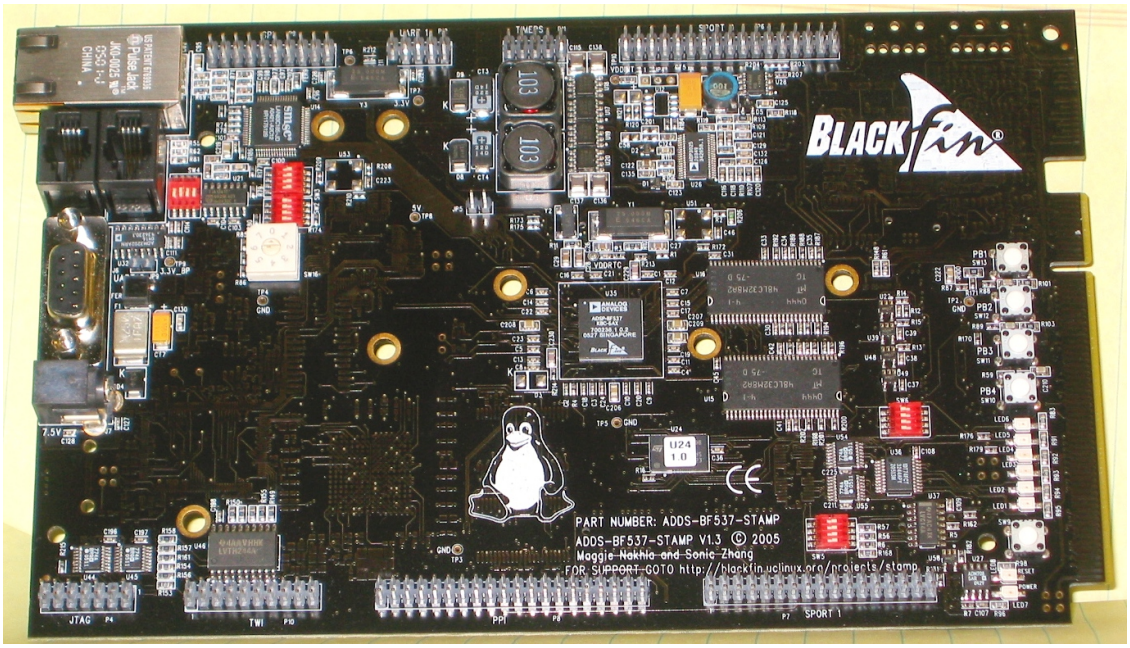

Figure 1: BF537-STAMP Board from Analog Devices

6.3 Compiling a kernel & Root Filesystem

First the developer uses the graphical configuration utility to select an appropriate Board Support Package (BSP) for his target hardware. Supported target platforms are STAMP for BF533, BF537, or the EZKIT for the Dual Core Blackfin BF561. Other Blackfin derivatives not listed like BF531, BF532, BF536, or BF534 are also supported but there isn't a default configuration file included.

After the default kernel is configured and successfully compiled, there is a full-featured Linux kernel and a filesystem image that can be downloaded and executed or flashed via NFS, tftp, or Kermit protocol onto the target hardware with the help of preinstalled u-boot boot loader. Once successful, further development can proceed.

6.4 Hello World

A further step could be the creation of a simple Hello World program.

Here is the program `hello.c` as simple as it can be:

```
#include <stdio.h>

int main () {
  printf("Hello World\n");
  return 0;
}
```

The first step is to cross compile `hello.c` on the development host PC:

```
host> bfin-uclinux-gcc -Wl,-elf2flt \
      hello.c -o hello
```

The output executable is `hello`.

When compiling programs that run on the target under the Linux kernel,

`bfin-uclinux-gcc` is the compiler used. Executables are linked against the uClibc runtime library. uClibc is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc. Library function calls like `printf()` invoke a system call, telling the operating system to print a string to stdout, the console. The `elf2flt` command line option tells the linker to generate a flat binary—`elf2flt` converts a fully linked ELF object file created by the toolchain into a binary flat (BFLT) file for use with uClinux.

The next step is to download `hello` to the target hardware. The are many ways to accomplish that. One convenient way is be to place `hello` into a NFS or SAMBA exported file share on the development host, while mounting the share form the target uClinux system. Other alternatives are placing `hello` in a web server's root directory and use the `wget` command on the target board. Or simply use ftp, tftp, or rcp to transfer the executable.

6.5 Debugging in uClinux

Debugging tools in the `hello` case are not a necessity, but as programs become more sophisticated, the availablilty of good debugging tools become a requirement.

Sometimes an application just terminates after being executed, without printing an appropriate error message. Reasons for this are almost infinite, but most of the time it can be traced back to something really simple, e.g. it can not open a file, device driver, etc.

`strace` is a debugging tool which prints out a trace of all the system calls made by a another program. System calls and signals are events that happen at the user/kernel interface. A close examination of this boundary is very useful for bug isolation, sanity checking, and attempting to capture race conditions.

If `strace` does not lead to a quick result, developers can follow the unspectacular way most Linux developers go using `printf` or `printk` to add debug statements in the code and recompile/rerun.

This method can be exhausting. The standard Linux GNU Debugger (GDB) with its graphical front-ends can be used instead to debug user applications. GDB supports single stepping, backtrace, breakpoints, watchpoints, etc. There are several options to have gdb connected to the gdbserver on the target board. Gdb can connect over Ethernet, Serial, or JTAG (rproxy). For debugging in the kernel space, for instance device drivers, developers can use the kgdb Blackfin patch for the gdb debugger.

If a target application does not work because of hidden inefficiencies, profiling is the key to success. OProfile is a system-wide profiler for Linux-based systems, capable of profiling all running code at low overhead. OProfile uses the hardware performance counters of the CPU to enable profiling of a variety of interesting statistics, also including basic time spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

The Blackfin gcc compiler has very favorable performance, a comparison with other gcc compilers can be found here: GCC Code-Size Benchmark Environment (CSiBE). But sometimes it might be necessary to do some hand optimization, to utilize all enhanced instruction capabilities a processor architecture provides. There are a few alternatives: Use Inline assembly, assembly macros, or C-callable assembly.

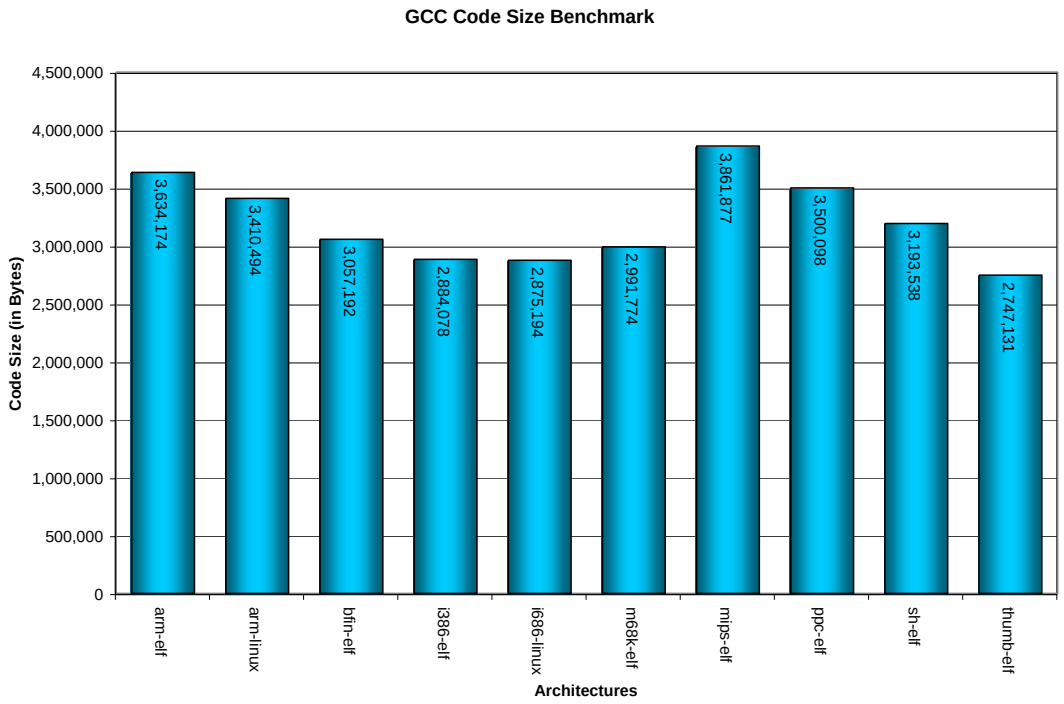

Figure 2: Results from GCC Code-Size Benchmark Environment (CSiBE) Department of Software Engineering, University of Szeged

6.6 C callable assembly

For a C program to be able to call an assembly function, the names of the function must be known to the C program. The function prototype is therefore declared as an external function.

```
extern int minimum(int,int);
```

In the assembly file, the same function name is used as the label at the jump address to which the function call branches. Names defined in C are used with a leading underscore. So the function is defined as follows.

```
.global _minimum;
_minimum:
   R0 = MIN(R0,R1);
   RTS;              /*Return*/
```

The function name must be declared using the `.global` directive in the assembly file to let the assembler and compiler know that its used by a another file. In this case registers R0 and R1 correspond to the first and second function parameter. The function return value is passed in R0. Developers should make themselves comfortable with the C runtime parameter passing model of the used architecture.

6.7 Floating Point & Fractional Floating Point

Since many low cost architectures (like Blackfin) do not include hardware floating point unit (FPU), floating point operations are emulated in software. The gcc compiler supports two variants of soft floating-point support. These variants are implemented in terms of two alternative emulation libraries, selected at compile time.

The two alternative emulation libraries are:

- The default IEEE-754 floating-point library: It is a strictly-conforming variant, which offers less performance, but includes all the input-checking that has been relaxed in the alternative library.
- The alternative fast floating-point library: It is a high-performance variant, which relaxes some of the IEEE rules in the interests of performance. This library assumes that its inputs will be value numbers, rather than Not-a-number values.

The selection of these libraries is controlled with the `-ffast-math` compiler option.

Luckily, most embedded applications do not use floating point.

However, many signal processing algorithms are performed using fractional arithmetic. Unfortunately, C does not have a fixed point fractional data type. However, fractional operations can be implemented in C using integer operations. Most fractional operations must be implemented in multiple steps, and therefore consume many C statements for a single operation, which makes them hard to implement on a general purpose processor. Signal processors directly support single cycle fractional and integer arithmetic, while fractional arithmetic is used for the actual signal processing operations and integer arithmetic is used for control operations such as memory address calculations, loop counters and control variables.

The numeric format in signed fractional notation makes sense to use in all kind of signal processing computations, because it is hard to overflow a fractional result, because multiplying a fraction by a fraction results in a smaller number, which is then either truncated or rounded. The highest full-scale positive fractional number is 0.99999, while the highest full-scale negative number is –1.0. To convert a fractional back to an integer number, the fractional must be multiplied by a scaling factor so the result will be always between $\pm 2^{N-1}$ for signed and 2^N for unsigned integers.

6.8 libraries

The standard uClinux distribution contains a rich set of available C libraries for compression, cryptography, and other purposes. (`openssl`, `libpcap`, `libldap`, `libm`, `libdes`, `libaes`, `zlib`, `libpng`, `libjpeg`, `ncurses`, etc.) The Blackfin/uClinux distribution additionally includes: `libaudio`, `libao`, `libSTL`, `flac`, `tremor`, `libid3tag`, `mpfr`, etc. Furthermore Blackfin/uClinux developers currently incorporate signal processing libraries into uClinux with highly optimized assembly functions to perform all kinds of common signal processing algorithms such as Convolution, FFT, DCT, and IIR/FIR Filters, with low MIPS overhead.

6.9 Application Development

The next step would be the development of the special applications for the target device or the porting of additional software. A lot of development can be done in shell scripts or languages like Perl or Python. Where C programming is mandatory, Linux, with its extraordinary support for protocols and device drivers, provides a powerful environment for the development of new applications.

Example: Interfacing a CMOS Camera Sensor

The Blackfin processor is a very I/O-balanced processor. This means it offers a variety of

high-speed serial and parallel peripheral interfaces. These interfaces are ideally designed in a way that they can be operated with very low or no-overhead impact to the processor core, leaving enough time for running the OS and processing the incoming or outgoing data. A Blackfin Processor as an example has multiple, flexible, and independent Direct Memory Access (DMA) controllers. DMA transfers can occur between the processor's internal memory and any of its DMA-capable peripherals. Additionally, DMA transfers can be performed between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller.

The Blackfin processor provides, besides other interfaces, a Parallel Peripheral Interface (PPI) that can connect directly to parallel D/A and A/D converters, ITU-R-601/656 video encoders and decoders, and other general-purpose peripherals, such as CMOS camera sensors. The PPI consists of a dedicated input clock pin, up to three frame synchronization pins, and up to 16 data pins.

Figure 3 is an example of how easily a CMOS imaging sensor can be wired to a Blackfin Processor, without the need of additional active hardware components.

Below is example code for a simple program that reads from a CMOS Camera Sensor, assuming a PPI driver is compiled into the kernel or loaded as a kernel module. There are two different PPI drivers available, a generic full-featured driver, supporting various PPI operation modes (`ppi.c`), and a simple PPI Frame Capture Driver (`adsp-ppifcd.c`). The latter is used here. The application opens the PPI device driver, performs some I/O controls (ioctls), setting the number of pixels per line and the number of lines to be captured. After the application invokes the `read` system call, the driver arms the DMA transfer. The start of a new frame is detected by the PPI peripheral, by monitoring the Line- and Frame-Valid strobes. A special correlation between the two signals indicates the start of frame, and kicks off the DMA transfer, capturing pixels-per-line times lines samples. The DMA engine stores the incoming samples at the address allocated by the application. After the transfer is finished, execution returns to the application. The image is then converted into the PNG (Portable Network Graphic) format, utilizing `libpng` included in the uClinux distribution. The converted image is then written to `stdout`. Assuming the compiled program executable is called `readimg`, a command line to execute the program, writing the converted output image to a file, can look like following:

```
root:~> readimg > /var/image.png
```

Example: Reading from a CMOS Camera Sensor

Audio, Video, and Still-image silicon products widely use an I2C-compatible Two Wire Interface (TWI) as a system configuration bus. The configuration bus allows a system master to gain access over device internal configuration registers such as brightness. Usually, I2C devices are controlled by a kernel driver. But it is also possible to access all devices on an adapter from user space, through the `/dev` interface. The following example shows how to write a value of 0x248 into register 9 of a I2C slave device identified by `I2C_DEVID`:

```
#define I2C_DEVID (0xB8>>1)
#define I2C_DEVICE "/dev/i2c-0"


i2c_write_register(I2C_DEVICE,
I2C_DEVID,9,0x0248);
```

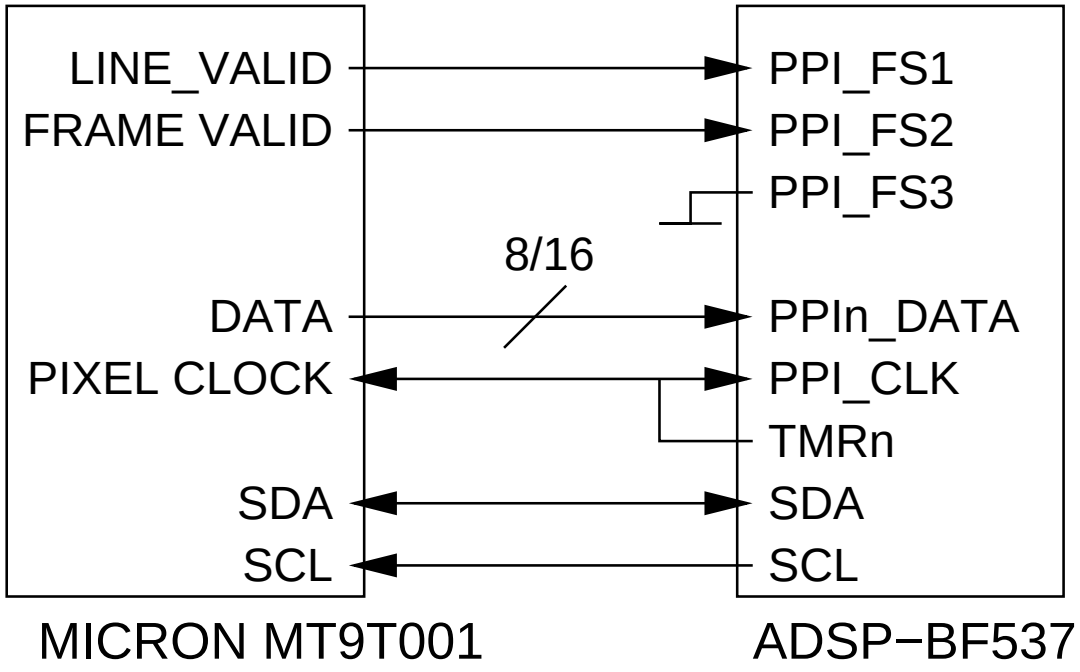

Figure 3: Micron CMOS Imager gluelessly connected to Blackfin

Example: Writing configuration data to e.g. a CMOS Camera Sensor

The power of Linux is the inexhaustible number of applications released under various open source licenses that can be cross compiled to run on the embedded uClinux system. Cross compiling can be sometimes a little bit tricky, that's why it is discussed here.

6.10 Cross compiling

Linux or UNIX is not a single platform, there is a wide range of choices. Most programs distributed as source code come with a so-called *configure* script. This is a shell script that must be run to recognize the current system configuration, so that the correct compiler switches, library paths, and tools will be used. When there isn't a configure script, the developer can manually modify the Makefile to add target-processor-specific changes, or can integrate it into the uClinux distribution. Detailed instructions can be found here. The configure script is usually a big script, and it takes quite a while to execute. When this script is created from recent autoconf releases, it will work for Blackfin/uClinux with minor or no modifications.

The configure shell script inside a source package can be executed for cross compilation using following command line:

```
host> CC='bfin-uclinux-gcc -O2 \
   -Wl,-elf2flt' ./configure   \
   --host=bfin-uclinux         \
   --build=i686-linux
```

Alternatively:

```
host> ./configure          \
  --host=bfin-uclinux      \
  --build=i686-linux       \
  LDFLAGS='-Wl,-elf2flt' \
  CFLAGS=-O2
```

```
#define WIDTH           1280
#define HEIGHT          1024

int main( int argc, char *argv[] ) {
    int fd;
    char * buffer;

    /* Allocate memory for the raw image */
    buffer = (char*) malloc (WIDTH * HEIGHT);

    /* Open /dev/ppi */
    fd = open("/dev/ppi0", O_RDONLY,0);
    if (fd == -1) {
        printf("Could not open dev\/ppi\n");
        free(buffer);
        exit(1);
    }

    ioctl(fd, CMD_PPI_SET_PIXELS_PER_LINE, WIDTH);
    ioctl(fd, CMD_PPI_SET_LINES_PER_FRAME, HEIGHT);

    /* Read the raw image data from the PPI */
    read(fd, buffer, WIDTH * HEIGHT);

    put_image_png (buffer, WIDTH, HEIGHT)

    close(fd);   /* Close PPI */
}

/*
 * convert image to png and write to stdout
 */
void put_image_png (char *image, int width, int height) {
    int y;
    char *p;
    png_infop info_ptr;

    png_structp png_ptr = png_create_write_struct (PNG_LIBPNG_VER_STRING,
                NULL, NULL, NULL);

    info_ptr = png_create_info_struct (png_ptr);

    png_init_io (png_ptr, stdout);

    png_set_IHDR (png_ptr, info_ptr, width, height,
                  8, PNG_COLOR_TYPE_GRAY, PNG_INTERLACE_NONE,
                  PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);

    png_write_info (png_ptr, info_ptr);
    p = image;

    for (y = 0; y < height; y++) {
        png_write_row (png_ptr, p);
        p += width;
    }
    png_write_end (png_ptr, info_ptr);
    png_destroy_write_struct (&png_ptr, &info_ptr);
}
```

Figure 4: read.c file listing

```
#define I2C_SLAVE_ADDR 0x38 /* Randomly picked */

int i2c_write_register(char * device, unsigned char client, unsigned char reg,
                       unsigned short value) {
    int    addr = I2C_SLAVE_ADDR;
    char   msg_data[32];
    struct i2c_msg msg = { addr, 0, 0, msg_data };
    struct i2c_rdwr_ioctl_data rdwr = { &msg, 1 };

    int fd,i;

    if ((fd = open(device, O_RDWR)) < 0) {
        fprintf(stderr, "Error: could not open %s\n", device);
        exit(1);
    }

    if (ioctl(fd, I2C_SLAVE, addr) < 0) {
        fprintf(stderr, "Error: could not bind address %x \n", addr);
    }

    msg.len   = 3;
    msg.flags = 0;
    msg_data[0] = reg;
    msg_data[2] = (0xFF & value);
    msg_data[1] = (value >> 8);
    msg.addr = client;

    if (ioctl(fd, I2C_RDWR, &rdwr) < 0) {
        fprintf(stderr, "Error: could not write \n");
    }

    close(fd);
    return 0;
}
```

Figure 5: Application to write configuration data to a CMOS Sensor

There are at least two causes able to stop the running script: some of the files used by the script are too old, or there are missing tools or libraries. If the supplied scripts are too old to execute properly for `bfin-uclinux`, or they don't recognize `bfin-uclinux` as a possible target, the developer will need to replace `config.sub` with a more recent version from e.g. an up-to-date gcc source directory. Only in very few cases cross compiling is not supported by the `configure.in` script manually written by the author and used by autoconf. In this case latter file can be modified to remove or change the failing test case.

7 Network Oscilloscope

The Network Oscilloscope Demo is one of the sample applications, besides the VoIP Linphone Application or the Networked Audio Player, included in the Blackfin/uClinux distribution. The purpose of the Network Oscilloscope Project is to demonstrate a simple remote GUI (Graphical User Interface) mechanism to share access and data distributed over a TCP/IP network. Furthermore, it demonstrates the integration of several open source projects and libraries as building blocks into single application. For instance `gnuplot`, a portable command-line driven interactive data file and function plotting utility, is used to generate graphical data plots, while `thttpd`, a CGI (Common Gateway Interface) capable

web server, is servicing incoming HTTP requests. CGI is typically used to generate dynamic webpages. It's a simple protocol to communicate between web forms and a specified program. A CGI script can be written in any language, including C/C++, that can read `stdin`, write to `stdout`, and read environment variables.

The Network Oscilloscope works as following. A remote web browser contacts the HTTP server running on uClinux where the CGI script resides, and asks it to run the program. Parameters from the HTML form such as sample frequency, trigger settings, and display options are passed to the program through the environment. The called program samples data from a externally connected Analog-to-Digital Converter (ADC) using a Linux device driver (`adsp-spiadc.c`). Incoming samples are preprocessed and stored in a file. The CGI program then starts gnuplot as a process and requests generation of a PNG or JPEG image based on the sampled data and form settings. The webserver takes the output of the CGI program and tunnels it through to the web browser. The web browser displays the output as an HTML page, including the generated image plot.

A simple C code routine can be used to supply data in response to a CGI request.

Example: Simple CGI Hello World application

8 Real-time capabilities of uClinux

Since Linux was originally developed for server and desktop usage, it has no hard real-time capabilities like most other operating systems of comparable complexity and size. Nevertheless, Linux and in particular, uClinux has excellent so-called *soft* real-time capabilities. This means that while Linux or uClinux cannot guarantee certain interrupt or scheduler latency compared with other operating systems of similar complexity, they show very favorable performance characteristics. If one needs a so-called hard real-time system that can guarantee scheduler or interrupt latency time, there are a few ways to achieve such a goal:

Provide the real-time capabilities in the form of an underlying minimal real-time kernel such as RT-Linux (`http://www.rtlinux.org`) or RTAI (`http://www.rtai.org`). Both solutions use a small real-time kernel that runs Linux as a real-time task with lower priority. Programs that need predictable real time are designed to run on the real-time kernel and are specially coded to do so. All other tasks and services run on top of the Linux kernel and can utilize everything that Linux can provide. This approach can guarantee deterministic interrupt latency while preserving the flexibility that Linux provides.

For the initial Blackfin port, included in Xenomai v2.1, the worst-case scheduling latency observed so far with userspace Xenomai threads on a Blackfin BF533 is slightly lower than 50 us under load, with an expected margin of improvement of 10–20 us, in the future.

Xenomai and RTAI use Adeos as a underlying Hardware Abstraction Layer (HAL). Adeos is a real time enabler for the Linux kernel. To this end, it enables multiple prioritized O/S domains to exist simultaneously on the same hardware, connected through an interrupt pipeline.

Xenomai as well as Adeos has been ported to the Blackfin architecture by Philippe Gerum who leads both projects. This development has been significantly sponsored by Openwide, a specialist in embedded and real time solutions for Linux.

Nevertheless in most cases, hard real-time is

not needed, particularly for consumer multimedia applications, in which the time constraints are dictated by the abilities of the user to recognize glitches in audio and video. Those physically detectable constraints that have to be met normally lie in the area of milliseconds, which is no big problem on fast chips like the Blackfin Processor. In Linux kernel 2.6.x, the new stable kernel release, those qualities have even been improved with the introduction of the new O(1) scheduler.

Figures below show the context switch time for a default Linux 2.6.x kernel running on Blackfin/uClinux:

Context Switch time was measured with `lat_ctx` from lmbench. The processes are connected in a ring of Unix pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. As number of processes increases, effect of cache is less. For 10 processes the average context switch time is 16.2us with a standard deviation of .58, 95% of time is under 17us.

9 Conclusion

Blackfin Processors offer a good price/performance ratio (800 MMAC @ 400 MHz for less than (US)$5/unit in quantities), advanced power management functions, and small mini-BGA packages. This represents a very low-power, cost- and space-efficient solution. The Blackfin's advanced DSP and multimedia capabilities qualify it not only for audio and video appliances, but also for all kinds of industrial, automotive, and communication devices. Development tools are well tested and documented, and include everything necessary to get started and successfully finished in time. Another advantage of the Blackfin Processor in combination with uClinux is the availability of a wide range of applications, drivers, libraries and protocols, often as open source or free software. In most cases, there is only basic cross compilation necessary to get that software up and running. Combine this with such invaluable tools as Perl, Python, MySQL, and PHP, and developers have the opportunity to develop even the most demanding feature-rich applications in a very short time frame, often with enough processing power left for future improvements and new features.

10 Legal

This work represents the view of the authors and does not necessarily represent the view of Analog Devices, Inc.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin