RTLINUX AND MICROBLAZE

Alejandro Lucero alucero@os3sl.com
www.os3sl.com

Version 0.2
23 March, 2007

1. Introduction
2. uClinux and Microblaze
3. RTLinux work
4. uClinux and RTLinux modifications
5. Results and Conclusions

1. Introduction

In 2003 John Williams began uClinux porting to the Microblaze soft processor[1]. With a Microblaze processor incorporated into the FPGA design, the developer can take advantage of using a operating system like uClinux, adding value to the final product, such as a full and standard network protocol stack.
In this document we present the work done to execute RTLinux, a hard-real time microkernel, on uClinux/Microblaze. Having a real time operating system adds even more flexibility since some parts can be done by software instead of hardware (FPGA configuration). The strength of a FPGA lies in the flexibility it introduces versus the ASIC design; working with an FPGA which has a microprocessor inside enables the execution of a broad code library; and, if you can execute code with real time requirements using the microprocessor, the flexibility goes one step further. This is another point developers will probably be grateful for (I guess).

This work was performed under OpenRTU project with funds from the Spanish Ministerio de Industria, Turismo y Comercio (FIT-330101-2004-5).

2. uClinux and Microblaze

As a low-cost processor, Microblaze lacks some functionalities that exist in high processors such as MMU (Memory Management Unit). With an MMU, an operating system can do a lot of valuable things in a easy way such as code execution with protection, swapping or code sharing between tasks. Without MMU, some of these are harder to achieve, such as code sharing (dynamic libraries), and others are impossible, such as swapping or protection. The trade-offs between MMU and noMMU will not be discussed here.

The GNU Linux operating system was designed with MMU in mind, so it is not possible to execute Linux on a processor without an MMU such as Microblaze. However, the uClinux project[2] began in 1996 with the aim of executing Linux in microcontrollers. The idea is to use all the software available with a general purpose/open source operating system (with the changes needed due to limitations), and to accept the drawbacks such as protection which can be controlled with a careful knowledge of what to execute and how to do it.

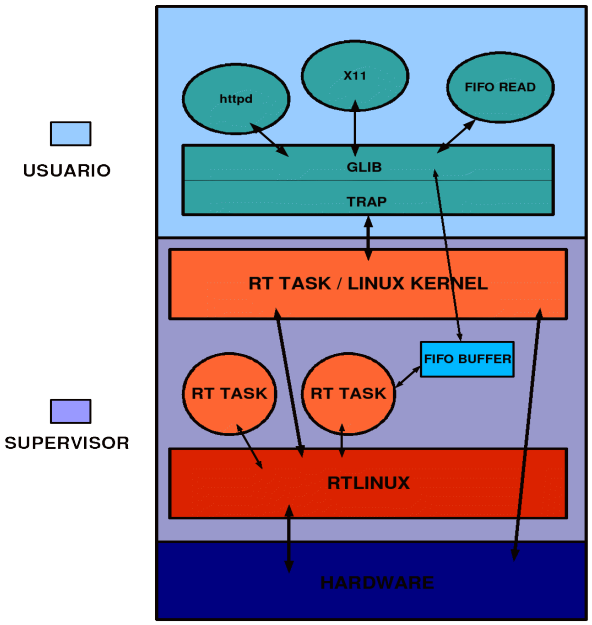
John Williams, from Queensland University, began uClinux porting to Microblaze in 2003 using a 2.4 uClinux distribution/kernel and currently it is stable enough to work with, although no official tests have been done to validate the work. Inside the OS it is necessary to test the stability of the parts related to the hardware: the usual Microblaze microcontroller configured to execute uClinux includes the processor core, the interrupt controller, the programmable timer, serial controller and ethernet controller . Other OS parts need not to be repeated since they were tested by the Linux Test Posix Suite Project[3] and are platform independent. However, although this architecture independence could exist, problems with the compiler could make some pieces of code faulty such as, for example, compiler optimizations modifying the instructions sequence. It would definitely be desirable to run all the full tests to find out what is the uClinux state on Microblaze.

During this project some tests using the Linux Test Posix Suite were carried out and the results were, in general, acceptable. The tests only covered the system calls, and the main problems appeared with protection faults, since these tests were thought to be executed with MMU and a usual test done with each system call is to pass a wrong pointer as a parameter. It is clear that more exhaustive tests are needed to evaluate the stability and the full LTP suite should be used. Our experience is that some parts are very stable, such as the network stack including service protocols such as NFS which was used for the RTLinux tests.

3. RTLinux work

Barabanov and Yodaiken[4] followed a new approach to obtain real time capabilities when using a general purpose operating system. Although the idea of virtualization had been used in other computer fields since the 70's, they were the first to apply it to real time systems. The concept consists of virtualizing the interrupts for the GPOS (Linux, BSD) and, with a POSIX compatible microkernel (RTCore) with real time functionalities,

taking control of the system. The full GPOS is seen as the lowest priority task from the RTLinux microkernel perspective.



The RTLinux approach is minimalistic, i.e., just the code needed for true real time will be in the microkernel. Communication between RT tasks and Linux processes is possible through RT FIFOS or shared memory, so data collected by an RT task can be stored on disk, copied to other systems with network communications or shown through a graphical interface. In this way the real time code remains small, a requirement for obtaining determinism in critical systems.

This approach solves the problem that traditional real time operating systems are facing right now: to maintain hard real time characteristics and at the same time offer functionalities demanded by developers, such as standard network protocols, databases, graphical interfaces, etc. On the other hand, the opposite happens when developers try to transform a GPOS such as Linux into an RTOS (the same system offering real time tasks and full GPOS functionalities). Although some improvements have been achieved following this approach, the results just guarantee soft real time and the future is not clear within this approach since Linux main design as a GPOS conflicts with real time requirements and, overall, the cost to get determinism is too high.

RTLinux runs in a broad range of processors and can be ported to any processor with a Linux version available. As a microkernel, RTLinux could be executed in non-Linux systems but the strength of RTLinux lies in this combination with Linux. The Linux/RTLinux relation implies that RTLinux works with MMU processors, BUT, this is not a requirement since RTLinux does not make use of MMU for its basic functionality. However, if MMU is present, RTLinux can take advantage of it offering additional options as the PSDD, a module which enables a RT task to be executed with memory protection mechanisms. The reason why RTLinux has not been ported[1] to non MMU processors is just commercial, since there are no clients interested in this kind of processor and hard real time/GPOS at the same time (until now?).

1 Some RTLinux ports to non MMU processors have been done but the sources have not been released.

OS3[5] has experience with RTLinux development, and some of the funders have contributed to some extensions of the RTLinux Free version in the past. The OS3 responsibility in the OpenRTU project was the RTLinux porting to the uCLinux/Microblaze architecture. This work was done in two phases:

1) Interrupts virtualization layer
2) RTLinux microkernel

We decided to divide the work clearly, doing tests after each phase was completed, due to the uncertainty of the technology used and the possible limitations of a soft processor. In case of problems with the full RTLinux microkernel coexisting with uClinux kernel, we could just make use of the virtualization mechanism for a simple system executing critical code when an event raises an interrupt without uClinux interference.

Our fears were justified (we believed) once we ran the first tests after the virtualization layer was implemented. This phase includes some patches to uClinux code and the basic virtualization mechanisms loaded as a uClinux dynamic module. The tests showed latencies that were too high when Microblaze uses the OPB bus accessing DRAM or SRAM memory. This bus is not processor-memory specific like the common LBP (Local Bus Processor) used in more powerful processors and it's used, at the same time, by internal peripherals like the serial port or the ethernet controller. The OPB configuration used allowed just one bus master, the Microblaze processor, so although the processor could suffer some delays if some slave was using the bus, its mastering ensures that this situation will not last long. It's worth mentioning, that these first tests were with the virtualization layer only and measuring the latencies of the timer interrupt. This is not an ideal test since the timer cannot be caught by a simple real time code because the timer is used for normal uClinux operation. Then, non real time code(uClinux) behavior could be interfering with the measures. To solve this problem we could use specific interrupts fired by external hardware such as an oscilloscope or a signal generator card, but at the moment we prefer to wait until the full RTLinux microkernel can be used to evaluate system performance.

The second part of the work was done to implement the full microkernel services: threads creation and destruction, scheduling, timer programming (one shot and periodic) and threads synchronization and communication. The RTLinux microkernel code is composed of architecture specific code (threads context switch, timer programming, threads initialization) which has to be modified, and by independent architecture (scheduling algorithm, posix communication, API calls) which will run without changes. The architecture specific part was implemented without any particular problem and just some limitations in the microprocessor functionalities delayed the work slightly. One is the lack of lock instructions that allow atomic operations like other processors have, for example the bit operation instructions: test_bit, clear_bit, set_bit, test_and_set, test_and_clear. With microblaze these instructions are implemented as functions executed with interrupts disabled. The problem is that once RT virtualization is running, these

functions (under uClinux domain) don't disable the interrupts by hardware: they just change some bits in software variables specifically used to do the virtualization. So, if we want to do atomic bit operations in RT code we need other functions doing real interrupt disabling, and in this situation it is easy to forget to use these new functions instead of the old ones in some parts (especially if you are reusing code). This problem caused some wrong measures which, in turn, caused the high latencies problem.
Another problem was the lack of some specific flags when the code was compiled, then some multiplications and divisions were being done by software instead of using the hardware units. The discovery of this problem was possible just by isolating the problematic code and looking at the assembler instructions. Another problem was the bad management of 64 bits operations by the 2.95 compiler used in the first part of the work. This forced us to renew the code base and download the last uClinux distribution available with the 3.4 gcc compiler. As well as solving the 64 bits operations problem, the new compiler showed a better performance when using the instructions reordering process to improve the pipelining. It's worth mentioning that the last uClinux distributions showed an improvement in some parts of the system, specifically with the ***proc*** virtual file system used during the development of the RTlinux porting.

Eventually, the latencies were acceptable, once the problems mentioned above were solved. However this was not as easy as it seems. It was not due to the problems themselves or the difficulty in finding them, but due to what we did (influenced by initial problems) when the system was not working as expected. Perhaps we suffered the uncertainty a soft processor has or we were just paying for this preconceived idea, but the problems were traced following the longest possible route: we did not doubt the implementation (initially), so the problem had to be the technology. On the positive side, the longest route was not a complete waste of time. As we show in the next section, we can make use of this work to obtain the best possible performance with Microblaze and RTLinux.

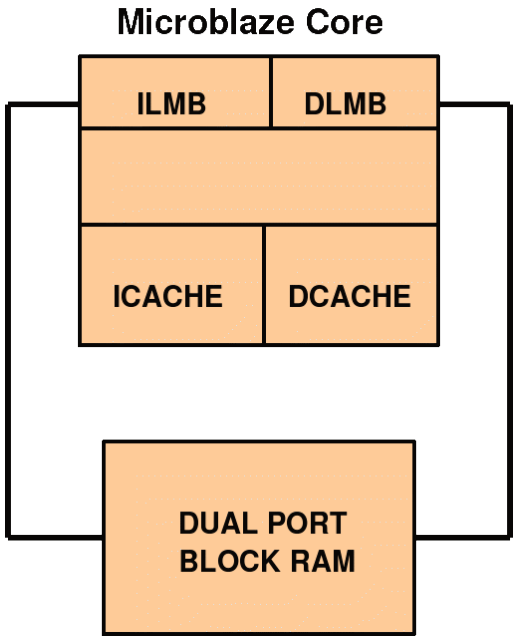
4. uClinux and RTLinux modifications

When the main RTLinux porting was done, the initial tests showed a low performance of the system. Under load stress, the interrupt routine associated with the timer interrupt had a good mean response but a worst case that was too high for the project goal, which has a requirement of 1ms (hard real time). The first hypothesis was that the OPB bus introduced some peak latencies under certain circumstances, which jeopardized the goal of determinism needed for hard real time. As mentioned above, as we had assumed the process lay with the technology, we took that to be the problem instead of considering possible bugs in the implementation[1]. The solution taken was to make use of a special Microblaze configuration avoiding the execution of real time code from DRAM or SRAM.

1 No doubt, we were influenced by a Microblaze design error we discovered at the initial phase of the project.

LMB and caches in Microblaze

The FPGA design provides two ways to execute critical code with Microblaze. The first uses a specific local memory bus (LMB) which has a latency access of one cycle to a specific internal memory. The second uses internal microprocessor caches (with one cycle access too) like other microprocessors, avoiding the microprocessor waiting for instructions or data and using algorithms based on temporal and locality reference principles. The idea was to allocate the real time code into these special memories to avoid the peaks shown when the code had to go through the OPB.



There's a problem using the LMB or Microblaze cache which must be taken into account: its use is not free and its size has an upper limit depending on the architecture. The implementation of LMB and microblaze caches is done with FPGA BRAMs, so its use can limit the implementation of DSPs or whatever is done within the FPGA logic. In our case, the project had different teams each working within its own area of expertise, and those people who were doing the DSP design did not know they had to share the BRAM with others.

It's also worth mentioning a point about using internal microblaze caches for real time code execution. The purpose of internal caches is to have inside them the code or data needed in the future. Of course, this is not always possible, but depending on the cache implementation and the code patterns, the cache will help to improve the system performance to some degree but with limitations. One advantage of a soft processor like Microblaze is that we can configure it to work as we need, and in this case, we can fix the addresses in the cache to guarantee that the code or the data[1] needed will always be allocated in the cache.

The RTLinux implementation is based on several modules, and the goal was to put all the code and data related to RTLinux into LMB and/or microblaze caches. Indeed, the first interrupt code related to uClinux(this is, since an interrupt is raised and the processor jumps to the 0x10 address until the function handle_irq is called), and which is used by

1 Data cache implementation is write through, so when data in the cache is updated the microprocessor waits until the data is updated in the external memory too.

RTLinux, should be in the one cycle memories too. And finally, the code of the real time thread/s to be executed should be allocated there.
The first problem was the LMB size: an upper limit of 16Kbytes for Spartan3 2000. However these are the sizes of the RTLinux modules:

- rtl.o: 8192 bytes (just code)
- rtl_time.o 2264 bytes (just code)
- rtl_sched.o 13692 bytes (just code)

Obviously, we had a problem since with these numbers the LMB was not big enough. Moreover, the usable size in LMB was only 12Kbytes if we wanted to follow the rules with the module insertion functionality available with uClinux[2]. And now that we have mentioned the module subsystem, how to force uClinux to use the specific addresses related to LMB or caches? We'll see after the advertisements! (read here, we'll see it shortly).
The first module, rtl.o, included the basic virtualization code and another one needed just for initialization, so what we did was to build another module which allocates just the code executed during the normal Rtlinux operation, leaving the initialization code and the code used for virtualization from uClinux perspective in the old modules. The new module was called rtl_previous core.o, and with a size of 8000 bytes it could be allocated into LMB.

At this point, with all the critical code and data allocated in one cycle access memories, we did new tests to evaluate the performance and, with a mixture of surprise and frustration, we discovered peak latencies had survived the attack. Then came a time when we began to doubt whether the code was correct, so the next step was to study the implementation. Some problems were not visible until we looked at the assembler code, which is not an excuse but makes us feel better.

Linux modifications

If we want to use specific code at specific addresses we need some way to do it. Rtlinux works with dynamic modules, so once the uClinux is running, the modules can be inserted when required providing the RTLinux core for real time requirements and doing the virtualization of interrupts in hot. Instead of this approach, Rtlinux code could be compiled with the uClinux code, then using the linker scripts we could put the special code at the special addresses in a static way. However, as we want to be loyal to Rtlinux design we needed another way, and, from a developer point of view it is more flexible to work with dynamic modules than to recompile the full kernel code when some change is done just to the RTLinux core. Moreover, although the RTLinux core could be compiled within the uCLinux kernel, the RT tasks need to be inserted after the RT core is working, so it is better (for flexibility) if we have a way to insert modules at specific addresses.

2 The memory allocated for a module must be page aligned, and the code for interrupts, exceptions, traps, etc, is allocated in the first addresses of the page 0 of LMB.

We thought of two possible solutions:

- To modify the insmod function allowing a new parameter for the desired addresses to load the module.
- A dirty modification of module.c file, controlling in some way the new module insertions and bypassing the uClinux memory management.

The first would be clear and in our opinion the best solution. The problem is the uClinux/Linux community is very restrictive when someone tries to change the system API, and with this approach changes must be done in the libc code and inside the uClinux kernel. Another important point is how uCLinux memory management must control these specific addresses. If we want to follow the rules with uClinux kernel internals, we should create new memory zones to differentiate between the normal memory and the LMB or the specific range used by the microblaze caches. Perhaps it does not make sense since these memories are going to be used for specific operations and they will not be used more than once by the system, and again, some previous considerations to add new memory zones in Linux have been underestimated before. Meanwhile, the second is faster and easier to implement, although we can not hope to be approved by the Linux purists.
Right now, we are doing the dirty modifications to the module.c file and the approach followed is very dirty: the control is done taking into account how many modules have been inserted into the system. This does not allow flexibility since if we discover that we need to insert a new module, we should modify the file and recompile the kernel. A less dirty solution would be to use the virtual proc file system to add module names with the desired addresses and the module.c file looking at this list to insert the modules.

Results and conclusions

Once the problems were resolved, the latencies were acceptable taking into account the architecture. Our long route to resolve the problems enables us to obtain the best results possible using the configuration options a soft processor such as microblaze has.

Periodic Task	*Microblaze Caches Enabled*	*Code at LMB?*	*System Load*	*Task jitter*	*irq timer worst case*
500us	Yes	No	IDLE	90	21
500us	Yes	No	Stressed	135	36
500us	Yes	Yes	IDLE	24	25
500us	Yes	Yes	Stressed	38	23
500us	No	No	IDLE	54	35
500us	No	No	Stressed	104	40
500us	No	Yes	IDLE	23	24
500us	No	Yes	Stressed	37	24
500us	Yes SRAM	No SRAM	IDLE	65	20
500us	Yes SRAM	No SRAM	Stressed	114	33

Our team had the responsibility to port RTLinux microkernel to the uCLinux/Microblaze combo. The hard real time requirement established in the design was 1ms needed to execute code related to a periodic event (hardware interrupt). The results show Rtlinux over uClinux/Microblaze can achieve these requirements, even without using special low-latency memories such as LMB or restricted-range internal caches. The system scalability is guaranteed allowing hard real time performance under 500us when these special memories are used to execute the critical real time code.
The work remaining is to carry out more exhaustive tests both from a uClinux perspective using the LTP suite and from a RTLinux perspective in terms of regression tests and debugger availability.

References

[1] http://www.itee.uq.edu.au/~jwilliams/mblaze-uclinux/
[2] www.uclinux.org
[3] http://posixtest.sourceforge.net/
[4]http://www.fsmlabs.com/a-linux-based-real-time-operating-system.html
[5] www.os3sl.com