Appeared in the Proceedings of the First Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Palma de Mallorca, July, 2005

Lightweight RTAI for DSPs

Jens Kretzschmar, Robert Baumgartl
Department of Computer Science
Chemnitz University of Technology, Germany
{*jens.kretzschmar, robert.baumgartl*}*@informatik.tu-chemnitz.de*

Abstract

Digital Signal Processors (DSP) are an interesting alternative to conventional microcontrollers for embedded real-time systems. Up to now, no open source operating system has been widely accepted for DSPs. The special architecture of these chips prevents a simple port of Linux and its real-time modifications.

This paper presents a novel approach to implement real-time APIs for DSP systems with small memory resources: remove Linux from the Real-Time Application Interface (RTAI). We realized this concept for the VLIW DSP platform TMS320C62x. The interdependencies between Linux and RTAI are identified and eliminated. We document necessary modifications to RTAI and present a methodology to systematically port RTAI modules to our system. The resulting operating system is compared and evaluated to origninal RTAI. A careful optimization results in an interrupt latency below 100 clock cycles. The system is small enough to fit into DSP internal memory.

1. Introduction

During the last couple of years, Linux has attracted much interest within the embedded system market for its obvious advantages: freedom, flexibility and performance. The advent of RTLinux [13] and RTAI [4] has provided the foundation for the construction of Linux-based hard real-time systems.

Embedded systems based on digital signal processors (DSP) have been omitted from this trend for two main reasons:

- scarce memory and
- non-existent competitive compilers.

Application and even more operating system programming has been a task for only a few very skilled specialists producing few lines of code per day. Operating systems for DSPs have been very efficient, small and expensive. Almost always they are closed source. Porting software across different DSP platforms is usually difficult due to non-standardized OS APIs.

This situation has changed somewhat with the introduction of Texas Instruments' C6x DSP family. This processor is able to address several megabytes of memory, it has a fairly orthogonal instruction set and the compiler produces reasonably optimized code. Not surprisingly, ports of uCLinux to the C6x DSP have been published [5, 6, 7].

With regard to the main application area of DSPs, it would be favorable to have not only the plain Linux kernel but its real-time extensions available. RTLinux applications could be executed on DSP platforms and it seems in general very attractive to have a well-established and open source real-time OS API available for DSPs. Up to now, neither RTAI nor RTLinux have been ported to a DSP architecture. Therefore, we decided trying to migrate RTAI to a DSP. For historical reasons, we solely concentrated on RTAI.

The port has been done for the TMS320C62x VLIW fixed-point DSP architecture [10]. The chip has a Harvard architecture and provides separate internal memory banks for program and data of 256 kBytes each. Depending on the actual hardware platform external RAM of different sizes are possible. Clock rates range from 150 to 300 MHz. Due to its downward compatibility, our solution is also usable for the more recent type TMS320C64x, which is also used in application-specific DSPs, e. g. the TMS320DM642 and DM643. Typically, these processors incorporate between 1 and 8 MBytes of internal RAM and are clocked between 400 and 1000 MHz.

There is also a floating-point variant of the architecture, the C67x, which has been omitted from our port so far due to certain architectural differences to the C62x. We feel it would be not too difficultto include that version in the port, should the need arise.

The target hardware platform is a so-called Imaging Evaluation Kit (IEK C64x) by the french company Ateme [1].

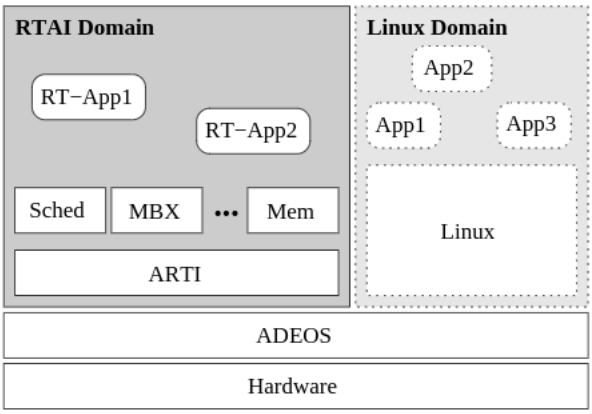

Figure 1. Basic architecture of RTAI

The rest of the paper is structured as follows. In section 2, we give a short overview of RTAI and the used development tools, motivate the elimination of Linux and give an overview of the porting process. Section 3 details some of the technical aspects and difficulties of the port. Efforts to optimize the resulting system are also described. The following section 4 documents the achieved performance and compares our solution to a conventional RTAI/x86 system. The final section 5 summarizes the main contributions of this work and gives a short outlook on further research.

2. Basic design

2.1. RTAI overview

Figure 1 depicts the basic structure of an RTAI system. The so-called ADEOS nanokernel realizes the domain concept and the virtual interrupt pipeline [12]. ADEOS is a separate entity and could also be used without RTAI, for instance to construct abstract machines. The ARTI (ADEOS Real-Time Interface) layer implements timers, generates `/proc` entries in Linux and contains some legacy functions. On top of it reside RTAI components which together constitute the real-time API and RTAI applications. From Linux' point-of-view both RTAI components and applications are specialized kernel modules. RTAI schedules the Linux domain (kernel and applications) with lowest priority, i. e. when no real-time application is ready to run. ADEOS prevents seizure of interrupts by Linux. More details on RTAI can be found elsewhere [4].

2.2. Compiler considerations

RTAI has been designed and implemented using the GNU Compiler Collection (GCC). Therefore, it would be natural to employ GCC for the port. Unfortunately, although GCC has indeed been ported (by our group) to the C6x DSP architecture, this port is not mature enough yet for serious development projects. Especially its optimizing features are inferior to Texas Instruments' C6x compiler [9]. Because our solution aims at low interrupt latency and low operating system overhead we decided against GCC.

For many years, Digital Signal Processors (DSPs) have been very special for programmers. To obtain optimum performance, it is quite common to implement even large software projects solely by means of hand-optimized assembler code. We did not pursue this approach for reasons of complexity and code maintainability. As we will demonstrate in section 4 this decision causes no intolerable performance penalty.

The only remaining option was to use the C compiler by Texas Instruments which represents the state-of-the-art in code optimization for the C6x architecture. Unfortunately, this tool is not open source. Additionally it is not fully compatible to GCC. For instance, the interface between C code and assembler statements differs in several ways from GCC.

By constantly improving our GCC port, we hope to migrate RTAI/C6x to that compiler in the future.

2.3. Eliminating Linux

To port RTAI to the DSP architecture, two principal design approaches can be distinguished. The first one is the 'classical': one of the Linux versions ported to the DSP must be extended with RTAI functionality.

Obviously, porting a monolithic kernel like Linux to a DSP causes serious overhead. Many kernel functions are obsolete within a DSP context (e. g. , drivers, memory management). Adding RTAI functionality (interrupt virtualization, real-time scheduler, . . .) to Linux is likely to increase that overhead. The resulting system will have a large memory footprint, large interrupt latency and large processing overhead. We believe this to result in an inacceptable performance penalty and therefore propose a radically different approach: Port RTAI to the DSP *without* the Linux kernel! Hopefully, the resulting system would be small and fast.

The feasibility of this approach strongly depends on the interdependencies between RTAI and Linux. If RTAI is using many Linux kernel functions, the port will be difficult, because these functions have to be reimplemented. As we will demonstrate, this is not the case.

In a typical RTAI system, Linux is used for

- booting the system,

- real-time system maintenance (adding tasks using `insmod`, modifying system behavior etc.),
- non-real-time communication to the RT subsystem (e. g. logging).

Additionally, it offers some advantages to let the development environment and the target platform reside on the same machine.

If these responsibilities are dispensable or can be emulated, the resulting system does not need Linux.

The Linux boot mechanism is not of any value for a DSP, because these devices usually boot from EEPROM or some interconnection network. System maintenance solely depends on the execution environment and therefore must be reimplemented anyway. Logging functionality is not immediately necessary and will be implemented if the need arises. Clearly it is impossible to use the DSP as development machine. Hence, we can conclude that Linux may be safely removed from an RTAI port to a DSP (the dotted parts in figure 1 vanish).

2.4. Challenges

Migrating software between such radically different architectures is obviously a demanding task. The peculiarities of the DSP and its C compiler rendered the port difficult but not impossible.

In contrast to conventional ports, two different types of functionality must be localized and reimplemented:

1. hardware-dependent code, and
2. functions provided by Linux.

The former comprises interrupt handling, context switch and timer manipulation. The latter is, fortunately, only a small set of functions: `printk()`, `kmalloc()`, `request_irq()`, some data type definitions and functions for bit manipulations (`bitops.h`). In addition, the Linux functions to transfer data between kernel and user address spaces (`copy_to/from_user()` have to be removed from certain RTAI modules (for instance, the mailboxes).

Hardware-dependent code in RTAI is well separated. The amount of porting work differs from module to module. The scheduler is—with the notable exception of the context switch—almost completely hardware- and Linux-independent. Memory allocation is a different story: the MALLOC module uses the Linux kernel function `kmalloc()`.

For rapid prototyping we used the following technique. Whenever a Linux kernel function was encountered, it was substituted by a reimplementation based on Texas Instruments' library without paying attention to memory overhead. For instance, a call to `kmalloc()` would simply be substituted by a call to the TI library function `MEM_alloc()`. After basic RTAI/C6x was up and running the according function was reevaluated, whether it could be safely removed, must be implemented from scratch or could be resubstituted by an RTAI module function which was not available initially. In the case of `kmalloc()`, the last option was chosen, that is, it was substituted by a call to `rt_malloc()` provided by the MALLOC module.

3. Porting RTAI

Porting RTAI to the C6x DSP by eliminating Linux can be structured as follows:

- adapting ADEOS,
- implementing ARTI functions (timers),
- migrating individual RTAI modules: scheduler, memory management, MBX, ...
- implementing a new module loader and
- optimizing the system.

The individual stages are briefly described in this section.

3.1. Adapting ADEOS

Porting the ADEOS abstraction layer was surprisingly easy. All functions and data structures that are important for registering and initializing a new domain were adopted. Simplification is possible, because ADEOS/C6x does not modify interrupt handling of operating systems executed on top. Functions that manipulate the interrupt hardware like `adeos_hw_cli()` were implemented from scratch. Many ADEOS functions are obsolete in the DSP context and have been subsequently omitted, among them

- ADEOS' semaphore implementation,
- SMP code,
- domain manipulation (suspending, deregistering).

3.2. Timer emulation

For obvious reasons, a very precise emulation of the RTAI timing functionality is crucial. Original RTAI time keeping uses three different facilities: the 8254 timer for generating interrupts, the time stamp counter (TSC) register for accurate execution time measurement and scheduling decisions and the Linux global kernel variable `jiffies`. The C6x provides three timers

which offer a reasonable resolution and are able to generate interrupts. Table 1 compares the timing facilities of the x86 and DSP architectures.

One of the DSP timers is used exclusively to provide an 8254 emulation. Another timer approximates the behavior of the TSC by using its shortest possible period. Due to the 40 bits precision of arithmetic operands, the maximum representable time value is 14660 seconds (four hours) for a 600 MHz DSP which might not suffice for all application scenarios. We hope to eliminate this limitation in the future.

The Linux variable `jiffies` is used very seldomly within RTAI. It would have been easy to replace it. Due to its usage at innumerable locations within the Linux kernel we felt it would be favorable to preserve it because very probably kernel (and device driver) programmers are used to its existence. To ease porting of RTAI/x86 applications, we decided to implement a `jiffies` emulation.

3.3. Porting RTAI modules

As mentioned in section 2.4 the scheduler is easy to port. Only the context switch is hardware-dependent because registers have to be saved and restored. Its implementation is straightforward though: all registers and the return address must be saved on the stack, certain task management structures must be manipulated (e. g. the pointer to the current task `rt_smp_current[0]` is updated), the stack pointer of the next task is obtained and finally the next task's registers have to be restored. Because the C6x is a uniprocessor the various multiprocessor schedulers were not regarded.

In original RTAI another context switch exists: the so-called domain switch is performed when control is transferred between different operating systems. In RTAI/C6x, domain switches never occur because it consists of only one domain. Therefore, domain switches have been eliminated.

The MALLOC module manages a memory heap available to applications by means of a buddy system. The size of this heap is defined at compile time. When RTAI is booted, the heap is allocated with a call to `kmalloc()` from Linux kernel space. When the heap is fully allocated by RTAI applications and more heap is still required it is extended by calling the internal function `alloc_extend()` which essentially allocates another chunk of Linux kernel memory via `kmalloc()`.

The following aspects of the module were modified. First, instead of calling `kmalloc()` at boot time, a memory block of the configured size is simply reserved by the development tools. Second, there is no way of increasing the size of the heap at run time (`kmalloc()` will always fail if the heap has been consumed). Hence, the programmer must carefully predict the needed heap size before. Finally, because the DSP offers two different memory spaces, MALLOC manages *two* heaps. The macros `rt_malloc()`, `rt_text_malloc()`, `rt_free()` and `rt_text_free()` constitute the programming interface.

Currently, only the most needed modules have been ported to RTAI/C6x: the scheduler, the memory allocator MALLOC, the semaphore SEM and two communication modules, MQ and MBX. A significant effort in the future will be to bring more modules into RTAI/C6x. The following methodology proved successful:

Redefine Data Types. RTAI extensively uses data types that are defined in Linux header files. Therefore, data types like `size_t` or `time_t` have to be redefined. Redefining data types was the main effort in porting both communication modules.

Including Linux Header Files. Every RTAI source file includes some Linux headers. In most cases, only very few pieces are really needed from these files. Either the required lines are brought into RTAI/C6x or the complete header file is taken over, as it has been done for `errno.h`.

Adapting RTAI Header Files. Every RTAI module defines its API in a separate header file which must be made known to RTAI by including it in `rtai_schedcore.h`.

Substituting Macros. All macros with variable parameter list must be substituted by an inline function due to limitations of TI's compiler. Other macros like `MODULE_LICENCE` etc. can be eliminated.

Reimplementing Linux functions. In RTAI/x86 modules may access all Linux kernel functions and data. Because Linux does not exist in RTAI/C6x, all these functions must be reimplemented or eliminated. Fortunately, in the modules ported so far only a small set of Linux functions and data was referenced. An example is `printk()` which is used extensively throughout RTAI for logging purposes. During debugging, this function is substituted by the TI library function `LOG_printf()` which passes the logging messages to the simulator. Another example is the `jiffies` variable for which we provide a suitable reimplementation as mentioned in section 3.2.

Handling 64-bits data. Operations on `long long` data (64 bits) are not supported by the compiler. The C6x DSPs are only able to operate on 40 bits wide operands. Therefore, it must be analyzed whether this reduced precision can be tolerated. If not, software emulation must be employed. The MSG module uses several 64 bits constants, therefore other communication

Table 1. Comparison of timing hardware in x86 and C6x architectures

	C6x, 600 MHz	x86 8254	x86 TSC, 600 MHz
register size	32 bits	16 bits	64 bits
frequency	75 MHz	1.19 MHz	600 MHz
shortest period	13 ns	840 ns	–
longest period	57 s	0.055 s	$3 \cdot 10^{10}$ s

modules were ported before.

3.4. Module loader

As a consequence of the elimination of Linux, the tools `insmod` and `rmmod` which are used in conventional RTAI systems to insert or remove individual modules and applications are not available anymore. A simple alternative is the transferral of one large binary file containing all RTAI modules and needed applications to the DSP at boot time. Because this prevents task and system modifications at runtime, we decided to implement a more flexible mechanism. Figure 2 illustrates its principle.

Providing a universal communication interface for all kinds of DSP hardware is beyond the scope of this paper, because DSP hardware is in general very heterogeneous (different communication interfaces, chip sets, external periphery ...). Therefore, we followed a pragmatic approach which solely concentrated onto our current hardware platform. Some implications for the implementation of a generic communication infrastructure between a Linux host and DSP hardware are described in [2].

In our current system configuration, the DSP communicates to the host processor via a JTAG interface. Due to lack of documentation, direct access to the host's JTAG interface is impossible. Instead, the Code Composer Studio IDE is used as a gateway. Our `insmod` and `rmmod` tools connect via COM to CCS which interacts with the JTAG port. The so-called RTDX libraries by Texas Instruments [11] encapsulate JTAG communication on both the DSP and the host side. We developed a very simple communication protocol between Host and DSP which is thoroughly described in [8]. Basically it allows to install and remove RTAI modules and applications.

Furthermore, the DSP needs a loader which analyzes the received executable files, relocates the contained sections and finally starts the program. Unfortunately, Linux' object loader could not be adapted for that purpose because it uses the ELF binary format, whereas the DSP compiler generates COFF binaries. Therefore, a new DSP module loader has been implemented from scratch. On arrival of a new COFF binary it does the following:

1. read section headers from received COFF file, allocate memory, copy sections into it,
2. relocate pointer destinations,
3. copy initial data values to the object's data memory,
4. register the module, save its exit handler,
5. jump to the entry symbol `_app_init`.

Module removal is straightforward. Several implementation problems and unclear technical documentation made the module loader one of the most difficult parts of our project. More details can be found in [8].

The loader is not needed if the task set is constant at runtime.

3.5. Optimization

After a timer interrupt event all software layers (IRQ handler, ADEOS, ARTI, scheduler) have to be traversed until finally a ready task is activated. Therefore, the time to process a timer interrupt was chosen as quantitative measure of the initial port's efficiency. The measurement was performed using a DSP simulator. Due to the C6x architecture such measurements are possible with single cycle precision.

Note that this time is not the interrupt latency (which is lower, cf. section 4.1).

The first RTAI/C6x version needed 1448 cycles for the activation of a task ready for execution. Almost half of the time was spent within ADEOS (cf. table 2). Therefore, ADEOS was analyzed and optimized as follows:

ADEOS is designed to manage several operating system instances in parallel. In a classical RTAI system two such domains are realized: RTAI itself and Linux. It is possible, to install and execute more operating system instances but to our knowledge this has not been

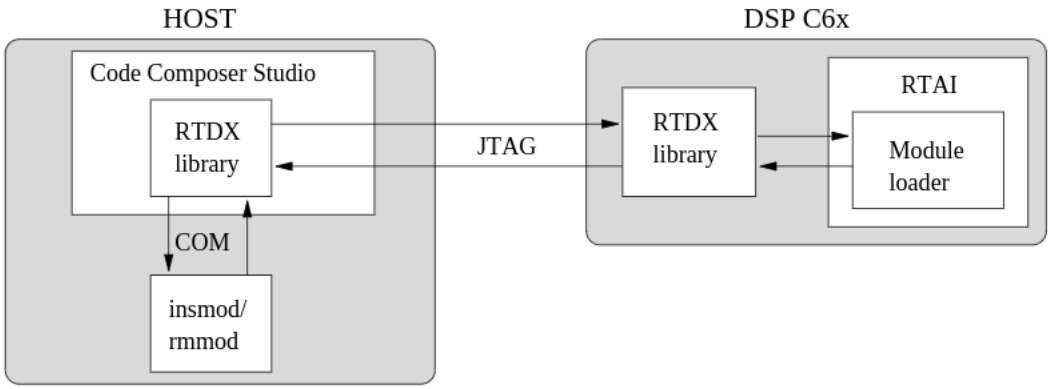

Figure 2. Communication between host and DSP

explored yet. Because we removed Linux from RTAI and we do not intend to execute another operating system it seemed reasonable to us to bypass the interrupt pipeline completely. As a consequence, several unnecessary functions were removed. The principle of queueing incoming interrupts in software when interrupts are disabled was not changed though in order to preserve standard ADEOS behavior. This optimization saved approximately 230 clock cycles.

Additional execution time improvements were achieved by exploiting certain architectural features of the DSP: inlining bit operations, optimizing interrupt enable and disable functions and parallel register saving. By this, we were able to save another 216 clock cycles.

Still, ADEOS needed some 320 cycles. Because low interrupt latency is crucial in many DSP applications we decided to introduce a change in interrupt semantics: only one incoming interrupt is queued up now if interrupts are disabled (interrupts may get lost when they are disabled for too long, now). Because critical sections are rare and short we feel that the limitation can be justified. This last optimization effort reduced ADEOS' execution time drastically to only 32 clock cycles!

4. Achieved performance

4.1. Interrupt latency

Two interrupt scenarios must be distinguished. An incoming timer interrupt causes all four RTAI components (IRQ handler, ADEOS, ARTI and the scheduler) to run, because it usually means that a task has been activated (cf. section 3.5).

When an other interrupt arrives, the IRQ handler, ADEOS and ARTI are executed whereas the scheduler is omitted, because the association of tasks with interrupts is performed in ARTI (no scheduling decision is necessary). Therefore, interrupt latency is the sum of the execution times of the IRQ handler, ADEOS and ARTI.

Table 2 shows the contribution of every RTAI component involved and reflects both interrupt latency and the time to process a timer interrupt. We compare our initial port, the current status obtained by the optimizations described in section 3.5 and a typical x86-based RTAI system (AMD Duron 650MHz, Linux 2.4.26, RTAI 3.1). All values are in clock cycles. The results for the x86-based system were obtained by reading out the Time Stamp Counter (TSC) register at selected points. We did not pay attention to serialization, therefore the results may be not as accurate as for the DSP.

The time to save registers has not been measured for the x86 version. Due to the low number of registers (nine), it can be neglected.

It it obvious that RTAI/x86 needs much more clock cycles due to the different chip architecture and the fact that original RTAI provides richer functionality.

The obtained results indicate that ADEOS/C6x is competitive. The optimization process proved successful. Especially the low interrupt latency renders RTAI/C6x suitable for reactive systems.

The optimization potential of the scheduler seems comparatively small as long as no change in its semantics is intended. A subject of further study could be the design of a specialized DSP scheduler which offers a similar performance boost as we obtained by optimizing ADEOS.

4.2. Task switch duration

Another important aspect is task switch efficiency (cf. Table 3). The duration of a task switch depends on whether the task to be activated has previously released the processor voluntarily (i. e. by

Table 2. Comparison of time to process a timer interrupt (in clock cycles)

	RTAI/C6x (initial)	RTAI/C6x (optimized)	RTAI/x86
IRQ Handler+saveregs	64	40	?
ADEOS	704	32	≈ 1182
ARTI	24	24	≈ 47
Scheduler	656	552	≈ 6804
interrupt latency	792	96	≈ 1229
timer interrupt processing	1448	648	≈ 8033

`rt_task_wait_period()`) or preemptively (has been interrupted by the timer). Again, we can report a significant performance gain by optimization.

Table 3. Task switch time (in clock cycles)

	RTAI/C6x (initial)	RTAI/C6x (optimized)	RTAI/x86
voluntarily	704	512	≈ 16800
preemptively	1448	784	≈ 11400

It is interesting to note that a x86 task switch needs 14 to 32 times more cycles than its C6x counterpart. This result is somewhat mitigated by the higher clock rates of x86-based processors but even considering the highest available clock rates for both processors the DSP version is in the lead.

More measurement results can be found in [8].

4.3. Memory footprint

One of the motivating factors for this project was to adapt RTAI for systems with very limited memory resources. Because the actual memory consumption is influenced by the number of needed RTAI modules and the number of application tasks, we consider two different systems.

A minimum system consisting of ADEOS, ARTI, SCHED and MALLOC requires only 39808 bytes of code and 6151 bytes of data memory. If the DSP BIOS by Texas Instruments (a kind of low-level library) is used, another 1216 bytes of code and 6816 bytes of data are needed. Every application task requires memory for its code, data, its stack and used application libraries and a certain amount of space for management structures (e. g. an `RT_TASK` structure).

The full set of ported functionality so far comprises the abstraction layers ADEOS and ARTI, the modules SCHED, MALLOC, SEM, MBX and MQ as well as the loader to dynamically modify the task set at run-time. This system requires 87104 bytes of code and 24474 bytes of data memory (plus the space for DSP BIOS, otherwise run-time communication to the host is impossible).

The numbers indicate that a real-time system based on Lightweight RTAI is very likely to fit into internal RAM of even the smallest C6x variant.

5. Conclusions and outlook

We demonstrate the feasibility of separating RTAI from Linux. Additionally, we show that it is possible to provide the RTAI API for a DSP without the overhead caused by the Linux kernel. Modules for scheduling, memory allocation, inter-process communication and semaphores have been ported to the DSP. Task switch time and scheduler performance are competitive in comparison to x86-based systems. Interrupt latency is especially low and outperforms x86-based systems by more than a factor of 10 in clock cycles. As expected, the memory footprint is small enough to locate RTAI and application tasks fully in DSP internal memory.

The main motivation for this work was to establish an open-source real-time API for the C6x DSP which allows manipulations of the task set at runtime. We feel this has been achieved.

Currently, the project is under active development. Some of the necessary improvements were mentioned throughout this paper. Among other, the following work must be done:

- porting remaining RTAI modules,
- implementation of DSP-specific RTAI modules,
- ease some of the module loader's limitations.

We will provide CVS snapshots at our website

http://rtg.informatik.tu-chemnitz.de

soon. Interested developers are welcome to join the project. Hopefully, ports to other DSP architectures will follow.

As soon as a competitive GCC for the C6x DSP is available, we will migrate our solution from Windows to the Linux host operating system. This project is currently under active development [3]. Several Linux drivers and tools for communication with DSP hardware have been developed. The GCC suite is improved continuously.

Another interesting question is whether Lightweight RTAI can be established for systems using conventional processors. This would require migrating drivers from Linux to RTAI which seems to be an interesting project itself!

References

[1] Ateme SA: *IEK C64x Imaging Evaluation Kit*, Data sheet, *http://www.ateme.com/products/iekc64.php*, 2005

[2] Robert Baumgartl, Ingo Oeser, Daniel Schreiber, Michael Schwindt: *DSP Accelerator Support for Linux*, Proceedings of the International Conference on Signal Processing Applications & Technology (ICSPAT), Dallas, October, 2000

[3] Robert Baumgartl, Ingo Oeser, Mirko Parthey, Adrian Strätling: *Bridging the Gap between Linux and DSPs*, Proceedings of European DSP Education and Research Workshop (EDERS), Birmingham, 2004

[4] Pierre Cloutier et al. *DIAPM-RTAI Position Paper*, Proceedings of the 21st IEEE Real-Time Systems Symposium and Real-Time Linux Workshop, Orlando, 2000

[5] Eatamar Drory, Or Sagi. *Introducing MediaLinux – A new real-time Linux Approach http://www.linux-devices.com/articles/AT554348551.html*, 2003

[6] Eatamar Drory. *Linux-programmable Audio-Video device based on TI DM64x DSP*, Proceedings of the Texas Instruments Developer Conference (TIDC), Birmingham, 2004

[7] Jaluna SA: *Jaluna OSware Linux Edition for TI C6000 DSP*, Product Datasheet, *http://www.jaluna.com/ fileadmin/pdf/TIDSP2305JalunaUS.pdf*, 2005

[8] Jens Kretzschmar. *Implementing RTAI on a DSP Processor without Linux*, Diploma Thesis, Chemnitz University of Technology, May, 2005

[9] Jan Parthey, Robert Baumgartl: *Porting GCC to the TMS320-C6000 DSP Architecture*, Proceedings of the Global Signal Processing Conference and Expo (GSPx), Santa Clara, September, 2004

[10] Texas Instruments, Inc.: *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000

[11] Texas Instruments, Inc.: *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*, November, 2001

[12] Karim Yaghmour. *Adaptive Domain Environment for Operating Systems*, Whitepaper, *http://www.opersys.com/ftp/pub/Adeos/adeos.pdf*, 2001

[13] Victor Yodaiken, Michael Barabanov. *Real-Time Linux*, Linux Journal, March, 1996