LP-ECOS: An Energy Efficient RTOS

Andrea Acquaviva, Tajana Simunic, Luca Benini
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2003-81
April 17th , 2003*

low-power, operating system, application driven

In this report we present an energy efficient implementation of a commercial-strength operating system (ECOS) running on a real-life hardware platform (HP SmartBadge IV). We integrate into the OS a power manager module that cooperates with the applications and power-aware device drivers through a set of APIs. The power manager makes decisions on power states of various system components, and sets processor clock frequency dynamically. We tested our RTOS implementation in single and multiple task environments. We used typical streaming multimedia applications that exploit wireless network and audio resources. Measurements obtained on MP3 audio decoder and other signal processing algorithms show that the whole system energy can be reduced by more than 50% with no effect on performance when using our extensions to the operating system.

* Internal Accession Date Only
Approved for External Publication

LP-ECOS: An Energy Efficient RTOS

Andrea Acquaviva Tajana Simunic Luca Benini

Abstract

In this report we present an energy efficient implementation of a commercial-strength operating system (ECOS) running on a real-life hardware platform (HP SmartBadge IV). We integrate into the OS a power manager module that cooperates with the applications and power-aware device drivers through a set of APIs. The power manager makes decisions on power states of various system components, and sets processor clock frequency dynamically. We tested our RTOS implementation in single and multiple task environments. We used typical streaming multimedia applications that exploit wireless network and audio resources. Measurements obtained on MP3 audio decoder and other signal processing algorithms show that the whole system energy can be reduced by more than 50% with no effect on performance when using our extensions to the operating system.

1. Introduction

Battery-operated portable systems impose tight constraints on the energy consumption. Low cost with fast time to market are critical. As a result, typical portable appliances have a microprocessor-based hardware architecture, which, coupled with embedded operating systems, eases application development. The energy efficiency of such systems can be improved by power-aware design of hardware and software, and by better utilization of system components at run-time.

In processor-based systems, such as palmtop computers and PDAs, the OS can play a strategic role in orchestrating power management for the entire system. A set of actions taken by the OS to configure processor and peripherals in order to reduce the system power consumption is called *power management policy*. Typical actions taken by the OS to reduce power include tuning processor voltage and frequency and switching among different power states available on both the processor and the other system devices. Only a few power-aware task scheduling and resource-allocation policies have been implemented in an operating system running on a hardware platforms in the past [9, 20, 21]. Unfortunately, none of the previous implementations integrate power management techniques for both the processor and the devices on a real hardware platform.

In this work we present a low-power version of Red Hat ECOS [1], a real-time embedded operating system. The new operating system, called LP-ECOS, supports both processor and peripheral power management. Processor clock scaling and run-time power state management is implemented by extending the standard hardware abstraction layer (HAL) of ECOS. Run-time device allocation/deallocation is possible because of OS collaboration with our power-enabled device drivers. Our power management strategy is application-driven. We designed a set of application-program interfaces (APIs), called LP-API, that collaborate with the OS-integrated power manager(PM) in order to: i) Efficiently manage processor power modes; ii) Perform dynamic clock frequency scaling, iii) Manage dynamic device shut-down and wake-up.

Following is an example of how LP-ECOS operates. The system normally starts with all components in the low-power mode. As soon as an application begins execution, the power manager is informed via LP-API call. Next, the application passes its utilization profile as a function of processor frequency to the PM. When such utilization profiles are available for all applications currently running in the system, the PM can set processor frequency to the minimum value guaranteed to meet performance constraints. During execution the application signifies to the power manager when it needs to access a resource, such as wireless network card. The PM updates a resource allocation table, and, if needed, wakes up the resource. In this way, the time overhead for device transition into the active states can be hidden due to application's internal initialization sequence. Once an application is done using a particular resource, the PM again updates the resource allocation table. If no other applications are using the particular resource, it is placed into a low-power state, thus saving energy. When an application terminates execution, it informs the PM. Once all applications have finished executing, the PM transitions all devices and the processor into the sleep mode.

Clearly, our approach gives the largest energy savings for systems with power-aware applications. Such applications use our LP-API to communicate with the power manager. Many portable systems come with a set of pre-loaded applications that can be very easily adapted to use our LP-API. In systems with an application mix involving both power-aware applications, and those that do not use our LP-API, the power manager can use predictive techniques such as those described in [18, 25].

In this report we show how LP-ECOS can obtain large energy savings because it has a tight collaboration between power-aware applications, power manager and power-aware device drivers. We ported LP-ECOS on the HP's SmartBadge IV [2] wearable platform running typical streaming multimedia applications such as the MP3 audio decoder and the front-end of a speech recognizer. The measurement results show that more than 50% of power savings can be obtained when applications exploit the new APIs to communicate processing and resource needs to the power

The rest of the report is organized as follows. Section 2 gives an overview of the related work. The implementation of LP-ECOS is presented in Section 3. We show measurement results with MP3 audio, speech recognition algorithm and low-pass filter running on the SmartBadge IV in Section 4. Finally, we summarize our findings in Section 5.

2. Related Work

A significant amount of work has been published in the context of system-level power management techniques. Power management strategies can be classified depending on what and how a low power hardware feature is exploited. Low power hardware features include dynamic power states configuration and run-time frequency/voltage adjustment [8, 29]. These features can be exploited using different policies.

A first major distinction can be made between policies aimed to shut-down components and those that scale down processing voltage and frequency dynamically. In [12] a predictive processor shut-down method using exponential average is presented in the context of general purpose systems. In [25] and [18] the predictive shut-down policy is based on Markov decision processes. Good results on power reduction are obtained with these policies on systems with large idleness. Run-time policy adaptation is presented in [3]. The cooperation between multimedia applications and the OS with the goal of controlling system resources to save energy have also been studied [7, 15, 16, 23].

Shutdown policies are not very effective in periods of extended system activity due to large power and performance cost incurred during state transitions. For such situations, Dynamic Voltage Scaling, DVS, can provide energy savings proportional to the square of the voltage. Many algorithms have been proposed that exploit this capability to save power 5, 6, 10, 11, 13, 14, 19, 22, 25, 27, 28].

Unfortunately, very small energy savings were measured when DVS heuristics have been implemented on real hardware, such as a part of Linux OS on Itsy portable device [9]. One of the main causes for such low power savings is very limited a priori knowledge of the application needs. On the contrary, work presented in [21] measures large processor power savings on DVS-enhanced hardware platform (LART) running Linux OS. The savings are largely due to the fact that information about the average task execution time, its start time and the deadlines is provided by the application to aid the OS in energy efficient scheduling.

A different approach is presented in [5] for a class of applications that know sufficiently ahead of time their processing characteristics and resource needs. The appropriate processor speed level is determined on-line by from a look-up table. The table contains the relationship between the minimum processor speed indexed by the important application processing characteristics. For example, when decoding the MP3 stream, both the sample rate and the bit rate are used as indicators of the computational effort of the processor.

In [20], the authors present a power-aware scheduling algorithm implemented on a DVS capable platform based on Intel XScale micro-architecture running ECOS operating system. The authors propose a set of APIs that can be used by tasks to communicate their characteristics (period, worst case execution time (WCET), deadline) to the OS. Using this information, the operating system can determine either statically or dynamically the processor frequency settings. Energy savings are estimated by determining the average power consumption of XScale instructions off-line for different frequency levels and then multiplying this value by the execution time.

In contrast to the previous work, this work presents a low-power RTOS operating system that for the first time integrates the support of all the three main approaches for lowering power consumption: (i) processor and peripheral shutdown (IDLE, SLEEP and OFF), (ii) processor frequency setting, (iii) application-driven DPM. In addition, we measured large energy savings on HP SmartBadge IV running typical multimedia applications when using LP-ECOS.

3. Low-Power Support

In this section we outline the low-power features implemented in our energy-efficient RTOS. We used ECOS real-time embedded operating system as our test vehicle. The structure of the power-aware version of ECOS, LP-ECOS, is shown in Figure [1]. It consists of four layers that span from the hardware to the application space. The first component is the Low-Power Hardware Abstraction Layer (LP-HAL), which defines architecture and platform specific modules. Kernel, which resides above HAL, consists of schedulers, support for thread synchronization, exception handling, interrupt handling and timers. Device drivers define mechanisms for accessing devices. The fourth component of LP-ECOS is the power manager. Based on the information obtained from the applications via LP-API, the power manager makes decisions on device power states (via power-aware device drivers) and processor power and performance states (via LP-HAL). The next Sections describe low-power enhancements in more details.

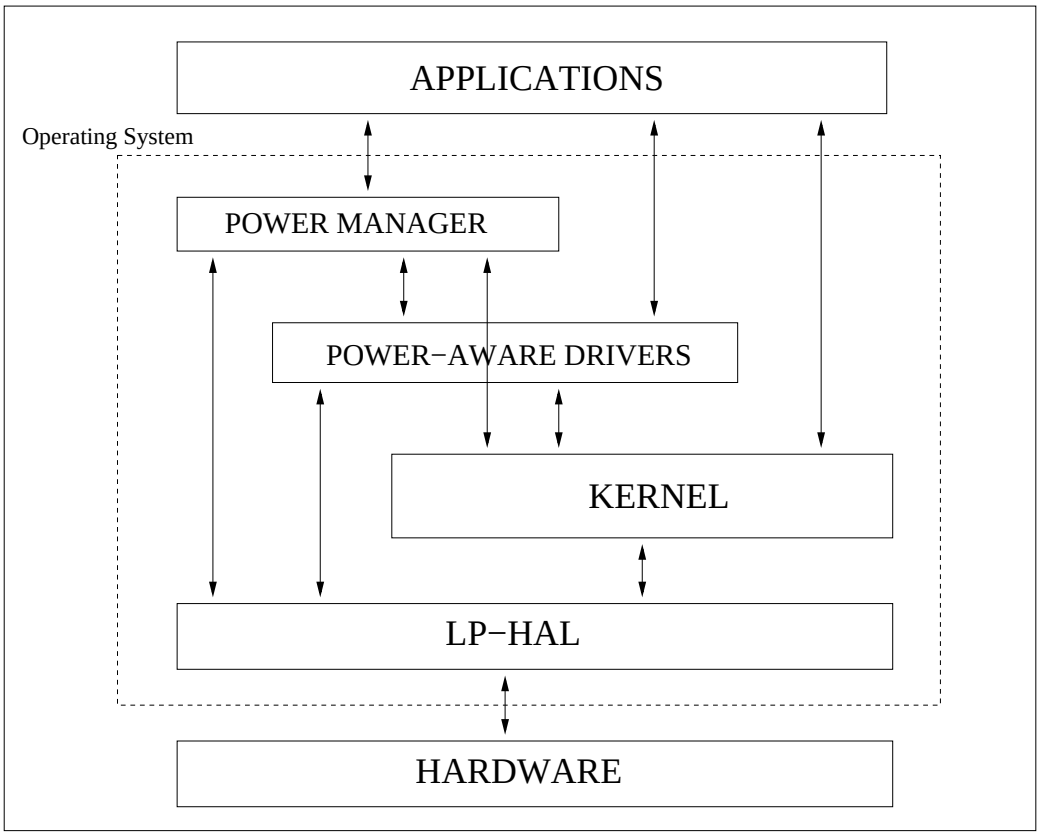

Figure 1. LP-ECOS System Architecture

3.1 Low-Power HAL

The hardware abstraction layer interfaces directly to hardware, thus hiding the architecture and platform details from the kernel. We modified ECOS HAL in order to control processor power modes and perform run-time clock scaling.

The StrongARM processor supports three different power modes: running, idle and sleep. The processor is placed in the idle state, in which the core clock is disabled, as soon as OS executes an idle thread. Instead of starting the idle thread, LP-ECOS executes the instruction sequence needed to put the processor into the idle state. The processor exits from the idle mode when the real-time clock triggers the timer interrupt routine. Since the period of the real-time clock is about 5ms, the overhead imposed by entering-exiting the idle mode, measured at 50us, is negligible.

Sleep state is more aggressive in terms of power reduction. When the processor sleeps, its power-supply is turned off, and thus the contents of all processor registers are lost. The processor can be woken-up only by pre-programmed external interrupt sources. The transition overhead to and from sleep is about 150ms. Power manager can reduce some of the overhead by saving only critical OS registers, and the registers related to devices currently in use. In addition to saving registers, the processor start-up routine

has been modified. The system initialization sequence contains a trap to distinguish between the execution of hardware reset and exit from the sleep state. In both cases the execution starts at the zeroth memory location. When waking up from the sleep state, the trap code gives the control of the execution to the PM. The PM restores the contents of the registers and re-enables the data cache. Note that the code controlling the sleep transition has been designed to support sleep state operation even during a remote debugging session.

We also implemented real-time clock scaling as a part of LP-HAL. Even though StrongARM's clock speed can be changed by writing into the PLL control register, a more complex sequence of instructions is needed to support dynamic frequency setting at run time. The frequency setting procedure changes depending on what devices are active at the time. Some on-chip units (LCD controller, DMA controller, OS timer, all serial controllers) experience an interruption for approximately 150us after writing to the PLL [24]. If one of these units is active, it must be disabled before changing the PLL configuration. The next step is to disable the caches and the clock switching, followed by changing the memory timing. Similar steps were taken in Linux OS implementation for Itsy [17].

In contrast with previous work, our frequency scaling implementation works in concert with the power manager. Thus we can speed up the clock transition by exploiting the knowledge PM has of which devices are currently in use. For example, if the on-chip audio controller uses DMA to perform input-output transfer, the PM saves DMA configuration registers and disables the DMA. When the new frequency is programmed, the DMA configuration is restored. If the DMA is not used by any device, this operation is not needed, so transition-time and energy can be saved.

3.2 Power-Aware Device Drivers

In this section we outline our implementation of device shut-down procedure as a part of the overall power management strategy. We illustrate the device shut-down methodology on two low-power device drivers we developed: wireless LAN card (WAVELAN) and audio CODEC.

The WAVELAN driver uses two dedicated functions, *wavelan_wake-up()* and *wavelan_sleep()*, to interface to the power manager. These functions save and resume the necessary registers of the WAVELAN card. In addition, the PCMCIA driver has been extended with two similar functions, *pcmcia_wake-up()* and *pcmcia_sleep()*. These two functions control the WAVELAN's power supply. With all four functions, the power manager is able to shut-down and restart the WAVELAN card in way completely transparent to the network stack.

Audio CODEC can also be shut-down by the power manager in a way similar to the WAVELAN. In addition, the audio driver implements another low-power feature aimed at saving the processor power. Since it is programmed to use the DMA controller in order to perform the data transfers, the processor is placed into the idle state when the DMA buffer is full and no other threads are active.

LP-APIs	Power Manager Interface
pm_application_start(app_handle, app_name) pm_application_end(app_handle, app_name) pm_res_alloc(app_handle, res_handle) pm_res_dealloc(app_handle, res_handle)	APPLICATIONS
pm_pcmcia_init(pcmcia_data) pm_wavelan_init(wavelan_data)	DRIVERS
pm_change_speed(new_speed)	APPLICATIONS & DRIVERS

Figure 2. LP-APIs

3.3 Power Management

The power management layer of LP-ECOS exploits the application knowledge via Low-Power API in order to more accurately control system resources, and thus lower the power consumption. Figure 2 lists the set of LP-APIs. When an application starts, it informs the power manager via *pm-application-start(app-handle, app-name)* function, where *app-name* is the name by which the application is identified, and *app-handle* is the associated handle. The PM in turn updates the application running list. As soon as an application either needs a resource or releases a resource, it informs the PM with a pair of function calls, *pm-res-alloc(app-handle, res-handle)* and *pm-res-dealloc(app-handle, res-handle)*, where *res-handle* is the handle associated to the resource. The PM responds by updating the resource application table and transitioning resources to/from low-power state. In this way, resources are started only shortly before the application actually needs them, and released as soon as they are not needed anymore. Often the time needed by the power manager to initialize a device can be masked by the initialization phase of the algorithm. Function *pm-application-end(app-handle)* is called once the application ends. When no more applications are running, the processor and all the devices can be put in the sleep state

As an example, we show in Figure 3 the pseudo-code of MP3 audio decoder illustrating LP-APIs. When the system boots, it initializes both the audio and the network interface. MP3 calls *pm-application-start()* function at its start time. For MP3 we implemented a frequency setting policy presented in [5]. This policy changes frequency at run time depending on the content of the header of the MP3 frames. As soon as the appropriate speed value is computed, the application calls the *pm-change-frequency(new-speed)* function to adjust the frequency. Shortly before first compressed frames are needed, the decoder calls *pm_alloc_res()* to inform the power manager of its need for the WAVELAN card. The power manager responds by waking up WAVELAN. Once the input buffer is filled, the decoder calls *pm_res_dealloc()* so that the power manager can shut-down the WAVELAN if it is not used by any other application. Then the collected frames are decoded and placed into an output buffer. Each time the output buffer is full, it is passed to the audio driver. Once the decoding is finished, the whole system transitions into sleep state.

```
pm_application_start(app_handle, app_name);
...
while(!end_of_bitstream)
{
      ...
      frequency_setting(bit-rate, sample-rate);
      ...
      pm_alloc_res(app_handle, res_handle);
      network_read()
      pm_dealloc_res(app_handle, res_handle);
      ...
      decode_collected_frames();
      ...
      output_decoded_frames();
 }

 pm_application_end(app_handle, app_name);
```

Figure 3. Pseudo-code of the MP3 decoder

4. Experimental Results

In this Section we present measurement results for LP-ECOS implementation on the HP SmartBadge IV device. We measure system power by monitoring the current absorption of the SmartBadge IV with National Instruments Data Acquisition (DAQ) board. The board is capable of collecting samples at a rate of 1.25 MHz, but we found that 1KHz was sufficient for most of our experiments. We tested our RTOS implementation in single and multiple task environments with typical streaming multimedia applications that exploit wireless network and audio resources.

4.1 Single-task Environment

We compare the power savings obtained when running MP3 audio decoder in a single-task environment on the SmartBadgeIV while using LP-ECOS. MP3 audio decoder receives frames through the wireless link, decodes them and plays the resulting audio out on the speakers. Table 1 shows the average power measured and the percentage of power savings obtained for four different test cases:

- *No policy*. Task runs at full speed with all devices always turned on.
- *CPU shut-down*. Task uses only processor shutdown but not device shutdown capabilities.
- *CPU & devices shut-down:* Both processor and device shutdown are exploited.
- *CPU & devices shut-down + clock scaling*. Processor clock frequency is scaled in addition to utilizing both processor and device shutdown. The frequency processor runs at is reported in the policy column.

The shut-down policies use our LP-HAL layer, power-aware device drivers and the implementation of LP-API in order to control both the processor and the device power states (WAVELAN and audio CODEC) . Processor transitions into running state as soon as MP3 decoder starts execution. MP3 uses calls to LP-API in order to inform the power manager of its device needs. The power manager responds

by initializing calls to power-aware WAVELAN and audio CODEC device drivers. When a resource is not needed during decoding time, the power manager is again informed via LP-API call, and responds by transitioning the resource to a low-power state. When decoding process is finished, the power manager first transitions all the devices to sleep through LP-API calls. The last step is to place the processor in a sleep state by initiating the shut-down sequence through LP-HAL layer.

Table 1. Power Consumption of MP3 Audio Decoder

Policy	P_{ave}(W)	% Reduction
No Policy	3.25	0%
CPU shutdown	2.69	17.2%
CPU & device shutdown	1.60	50.7%
CPU & device shutdown @191.7 MHz	1.49	54.1%
CPU & device shutdown @176 MHz	1.35	58.4%

In addition to shut-down policies, the MP3 decoder also utilizes frequency scaling via LP-API calls. The performance of the MP3 decoding process is a function of bit and sample rates, both of which are available in the MP3 header. Thus, the minimum clock frequency that provides the required decoding frame rate can be obtained based on the pre-stored curves. The curves show a trade-off between the frame-rate versus processor frequency for different bit and sample rates. This technique is described in more details in [5]. At the start of the MP3 decoding process, LP-API function call provides the power manager with the MP3 frequency-performance trade off curves. The PM responds by calculating the minimum clock frequency, and then calls the LP-HAL layer to actually perform the processor frequency adjustment. This process is repeated when either bit or sample rate change.

The results in Table 1 show that the combination of processor clock scaling, device and processor shut-down more than double the system power efficiency. About 50.7% of power savings occur when only processor and device shut-down are utilized. In addition to saving power, the performance improves because the CPU does not process useless broadcast packets coming from the network since the WAVELAN card is shut down as soon as incoming frame buffer is full. Additional 8% savings are obtained by using processor clock scaling. Such small increase in power savings is due to two issues. First, the MP3 decoder is computationally quite intensive, so as a consequence the clock speed cannot be safely decreased under 176MHz. Second, we use only frequency scaling. We could obtain much larger power savings if we also scaled down the voltage while slowing down the frequency. Additional hardware needs to be built to enable voltage scaling on the SmartBadge IV.

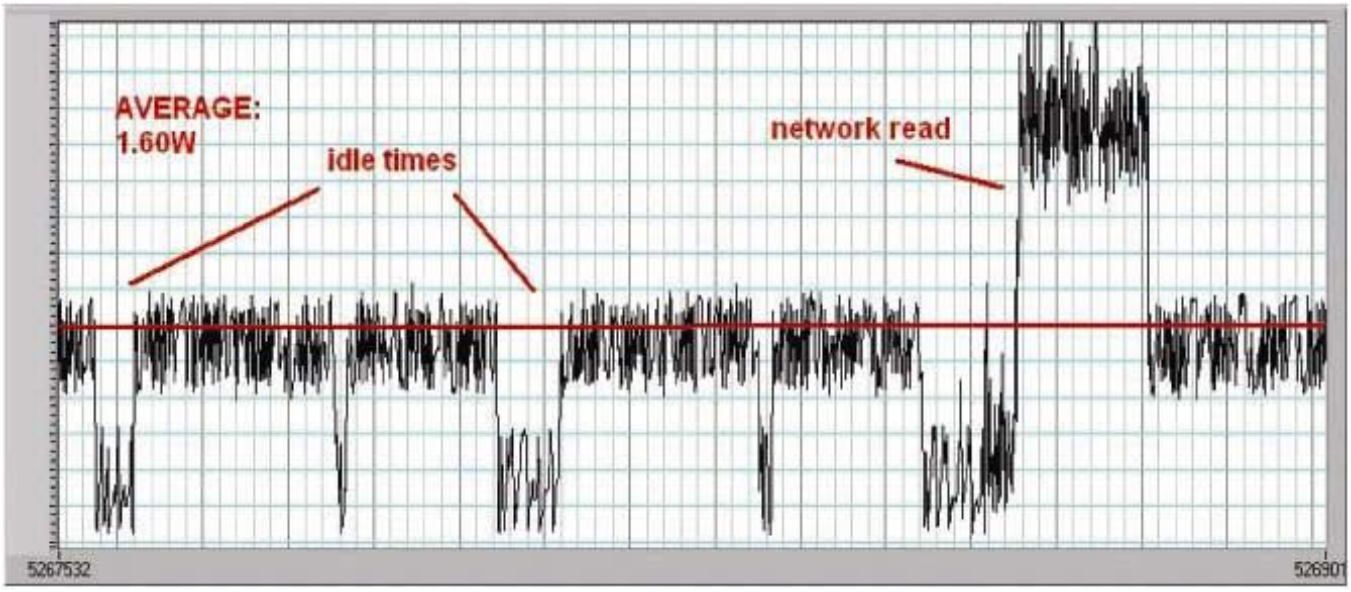

Figure 4. MP3 current at 202.6 MHz

In addition to reporting the average power measurements, we also show the current waveforms corresponding to MP3 execution at full speed, in Figure 4, and at reduced speed, in Figure 5. The peak current consumption can be observed in both Figures when the WAVELAN card is activated by the decoder in order to perform network reads. Since MP3 decoder runs faster than real-time at full speed, we can see the drop in current levels when the processor transitions into the idle state (Figure 4). In contrast, when running at reduced speed (Figure 5), there are no chances to place the processor into the idle state since now MP3 decodes close to real-time.

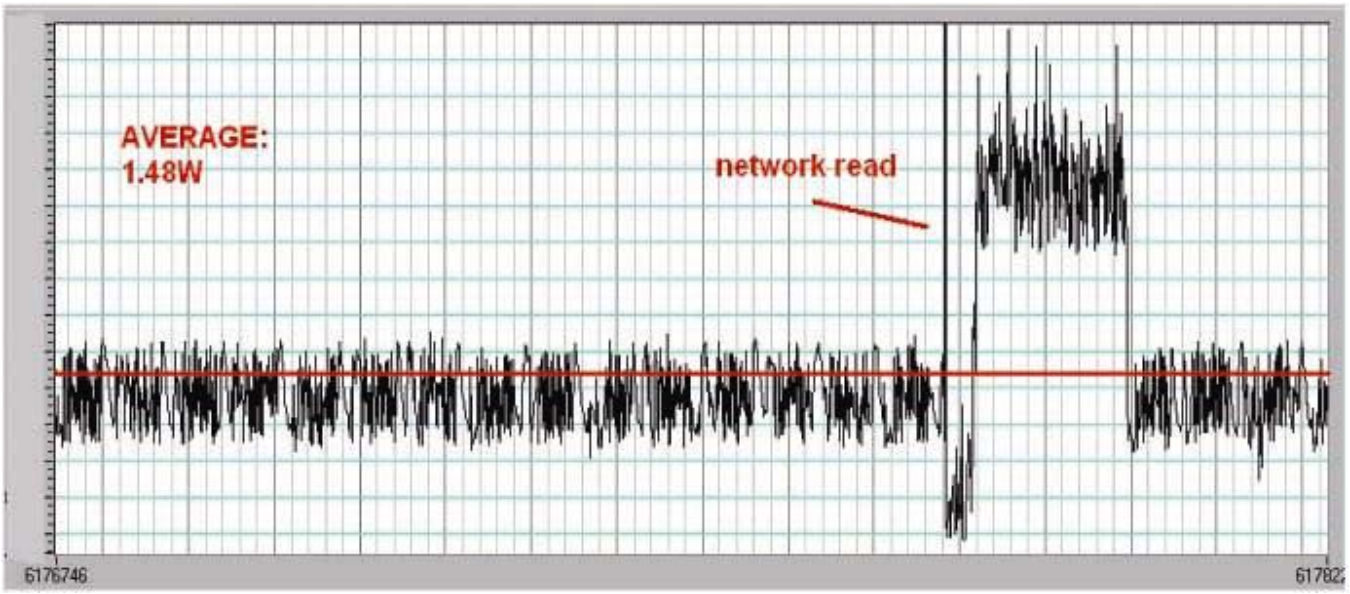

Figure 5. MP3 current at 191.7 MHz

4.2 Multi-Task Environment

LP-ECOS is well suited to handle multi-task environment. In this Section we show the measurement results when running two different streaming applications on SmartBadge IV. In our test case, we use a low-pass filter and a front-end of a speech recognizer [4]. The first application reads data through the wireless network and sends it to the audio output, while the second task takes speech samples from the audio input and sends them to a server via network interface.

The utilization versus frequency characteristics for both applications are obtained off-line, and are then supplied to the power manager via LP-API calls. Let us assume that the filter application starts first. The power manager in turn sets the processor frequency to the minimum value that keeps the processor utilization less than 100% (to take into account intertask interference). The value is obtained from pre-stored utilization characteristic, such as the one shown in Figure 6. As soon as the speech recognizer starts, the power manager recalculates the minimum allowed processor frequency setting. The new setting is a value of processor frequency for which the sum of utilizations of both applications is less then 100% (see the top curve in Figure 6).

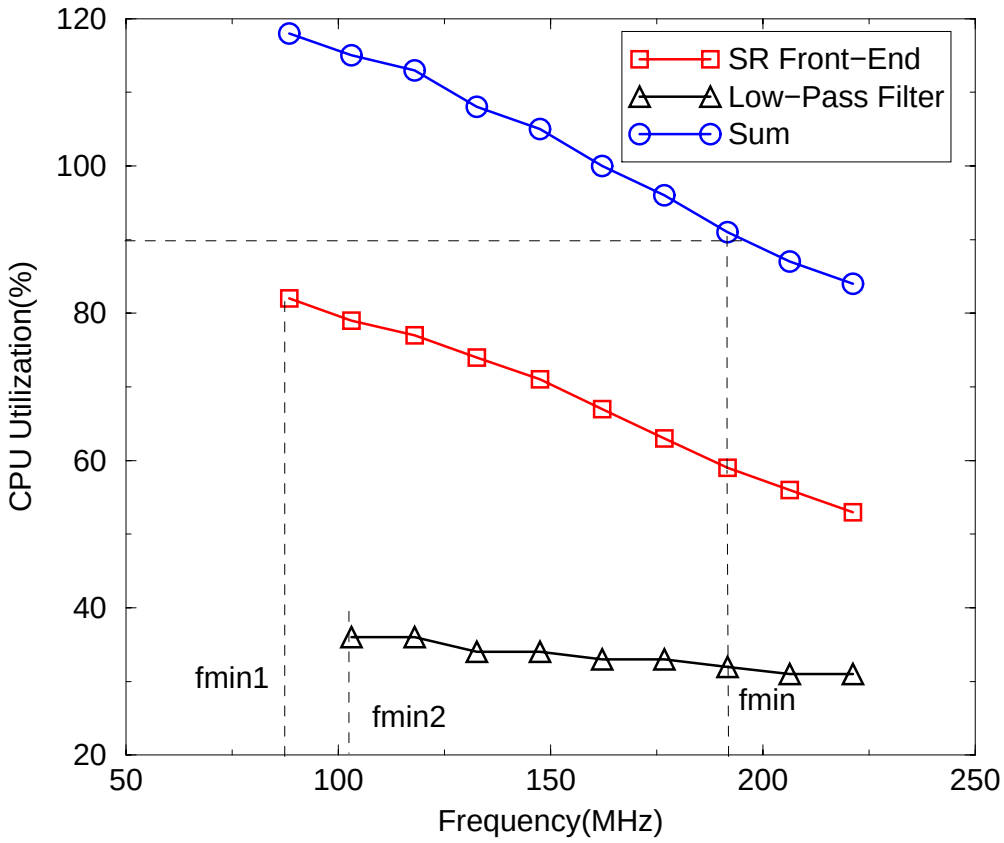

Figure 6. Characterization of streaming applications

In addition to processor frequency scaling, each application communicates its resource requirements through LP-APIs. In this way the power manager can allocate and deallocate I/O devices on demand. A device transitions to the active state as soon as the power manager receives a first request from an application for that device. When all applications release a resource, the power manager places the device into the sleep state. Once all tasks in the system terminate, the power manager puts the CPU into the sleep state as well.

Table 2 shows that the processor and device shut-down save 25% of power when running in a multi-task environment described here. Adjusting the processor frequency leads to additional 10% power savings with respect to the CPU and device shut-down policy. As discussed earlier, the processor frequency scaling savings are smaller since they do not include the voltage scaling. Adding voltage scaling would significantly increase the overall power savings.

Table 2. Energy Consumption in Multi-Task Environment

Policy	P_{ave}(W)	% Reduction
No Policy	2.83	0%
CPU & device shutdown	2.10	25.7%
CPU & device shutdown @191.7 MHz	1.84	34.9%

5. Conclusions

In this work we presented a complete implementation of an energy-efficient operating system, LP-ECOS. LP-ECOS cooperates with the applications when making both processor and peripheral power management decisions. We illustrate our ideas by running typical streaming media applications on the HP SmartBadge IV in cooperation with LP-ECOS. The experimental results show that up to 58% energy savings when running MP3 audio decoder with our approach. Similarly, we obtain more than 34% savings when running two streaming media applications concurrently.

6. References

[1] http://sources.redhat.com/ecos/

[2] G. Q. Maguire, M. Smith and H. W. Peter Beadle, "SmartBadges: a wearable computer and communication system", *6th International Workshop on Hardware/Software Codesign*, 1998.

[3] E. Chung, L. Benini and G. De Micheli, "Dynamic Power Management for non-stationary service requests", *Proceedings of DATE*, pp.77--81, 1999.

[4] B. Delaney, N. Jayant, M. Hans, T. Simunic, A. Acquaviva, "A Low-Power, Fixed-Point, Front-End Feature Extraction for a Distributed Speech Recognition System", *IEEE ICASSP*, May 2002.

[5] A. Acquaviva, L. Benini, B. Ricco, "Software Controlled Processor Speed Setting for Low-Power Streaming Multimedia," *IEEE Transactions on CAD*, pp. 1283--1292, November 2001.

[6] A. Sinha, A. Chandrakasan, "Energy efficient real-time scheduling," *ICCAD*, pp. 458--463, Nov 2001.

[7] F. Bellosa, "Endurix: OS-Direct Throttling of Processor Activity for Dynamic Power Management," *Technical Report TR-I4-99-03*, University of Erlangen, June 1999.

[8] L. Geppert, T. Perry, "Transmeta's magic show," *IEEE Spectrum*, vol. 37, pp. 26--33, May 2000.

[9] M. Neufeld, D. Grunwald, P. Levis, C. Morrey, K. Farkas, "Policies for Dynamic Clock Scheduling," *OSDI*, San Diego, USA, October 2000.

[10] K. Govil, E. Chan, H. Wasserman, "Comparing Algorithm for Dynamic Speed-Setting of a Low-Power CPU,"*International Conference on Mobile Computing and Networking*, pp. 13--25, November 1995.

[11] I. Hong, M. Potkonjak, M. B. Srivastava, "On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processors," *ICCAD*, pp. 653--656, November 1998.

[12] C. H. Hwang, A. C. H. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-Driven Computation," *ICCAD*, pp. 28--32, November 1997.

[13] T. Ishihara, H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processor," *ISLPED*, pp. 197--202, August 1998.

[14] S. Lee, T. Sakurai, "Run-time voltage hopping for low-power real-time systems," *ISLPED*, pp.806--809, July 2000.

[15] J. Lorch, A. J. Smith, "Software Strategies for Portable Computer Energy Management," *IEEE Personal Communications*, pp 60--73, June 1998.

[16] Y. Lu, L. Benini, G. De Micheli, "Operating System Directed Power Reduction," *ISLPED*, pp 37--42, July 2000.

[17] T. L. Martin. "Balancing Batteries, Power and Performance: System Issues in CPU Speed-Setting for Mobile Computing," *PhD thesis*, Carnegie Mellon University, 1999.

[18] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes", *Design Automation Conference*, pp. 555--561, 1999.

[19] T. Pering, T. Burd, R. Brodersen, "Voltage Scheduling in the lpARM Microprocessor System," *ISLPED*, July 2000.

[20] C. Pereira, V. Raghunathan, S. Guptha, R. Ghupta, M. Srivastava, "A Software Architecture for Building Power Aware Real Time Operating Systems," *ISLPED*, 2002.

[21] J. Pouwelse, K. Langendoen,H. Sips, "Energy Priority Scheduling for Variable Voltage Processors," *ISLPED*, pp. 28--33, Huntington Beach, USA 2001.

[22] Y. Shin, K. Choi, "Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems," *DAC*, pp. 134--139, June 1999.

[23] J. Flinn, M. Satyanarayanan, "Energy-aware adaptation for mobile applications," *In Proceedings of SOSP*, December 1999.

[24] Intel, "Intel StrongARM SA-1110 Microprocessor Advanced Developer's Manual," June 2000.

[25] T. Simunic, L. Benini, P. Glynn, G. De Micheli, "Event-driven Power Management," *IEEE Transactions on CAD*, July 2001.

[26] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, G. De Micheli, "Dynamic Power Management of Portable Systems," *IEEE Transactions on Computer-Aided Design*, July 2001.

[27] I. Weiser, B. Welch, A. Demers, S. Shenker, "Scheduling for Reduced CPU Energy," *SOSDI*, pp. 13--23, November 1994.

[28] F. Yao, A. Demers, S. Shenker, "A Scheduling Model for Reduced CPU Energy," *Annual Foundation of Computer Science*, pp. 374--382,

[29] http://developer.intel.com/design/intelxscale/

Appendix A

Installation and testing of LP-eCos on SmartBadge IV

This appendix describes how to install, configure and run a demo of a multimedia streaming application on LP-eCos on the SmartBadgeIV using Windows XP as a host and National Instruments DAQPad6070E as a data acquisition board.

1. Introduction

The following software is needed on the Windows XP host system for installation of LP-eCos on the SmartBadge IV:

- LP-eCos distribution
- Cygwin software under Windows
- GNU Tools
- Apple AirBaseStationConfig

And the following additional hardware:

- SmartBadgeIV with Lucent Wavelan Card
- Ethernet card
- Java Run-Time environment
- Apple Airport Base Station

DAQpad6070E board (or similar) and NI – DAQ Data Acquisition Driver Software for Windows can be used to perform power measurements.

2. Host software setup

The software configuration for the host system includes these four steps:

a. Downloading and installing Cygwin environment
b. Downloading and building the GNU tool chain
c. Unpacking and configuring the LP-eCos distribution
d. Build and download RedBoot Rom Monitor
e. Build, download and run an example program

2.a Downloading and installing Cygwin

Cygwin is a tool that provide a linux-like environment under windows. We need it in order to use the GNU toolchain to compile and build eCos.

To install the Cygwin net release, go to http://cygwin.com/ and click on "Install Cygwin Now!". This will download a GUI installer called **setup.exe** which can be run to download a complete cygwin installation via the internet. Follow the instructions on each screen to install Cygwin.

If you want to directly install the necessary packages from the net, choose "Install from Internet". You will be required to specify the root installation directory and the packages directory. Then specify how you want it to connect to the internet. If you are behind a proxy, type the name and the proxy port like:

- Use HTTP/FTP Proxy:

  Proxy Host: proxy.hpl.hp.com

  Port: 8088

At that point you will be prompted for choosing the packages to be installed. The basic packages you need are:

- Base
- Devel
- Shells → bash
- System → setup
- Utils → bzip2, cygutils, patch

Now download the selected packages. Each time you need to upgrade the packages, run the set – up utility and select the packages you want to add. If the installation procedure ends successfully, you will find a shortcut to cygwin bash shell on your desktop.

2.b Downloading and building the GNU tool chain

eCos requires arm-elf development tools to develop programs for ARM targets. The development tools come in three parts: the GNU compiler collection (GCC), the GNU Debugger (GDB) and the GNU binary utilities, which includes the GNU assembler and linker.

You can find a detailed explanation of this step in the Red Hat eCos web page: "**Building the ARM development tools for Windows**" at http://sources.redhat.com/ecos/tools/win-arm-elf.html.
In this page you will find the related link to the proper download pages. Otherwise, try directly with the following links:

- Binary Utilies:
  ftp://mirrors.kernel.org/gnu/binutils/binutils-2.10.1.tar.gz

- C and C++ compilers:
  GCC 2.95.2 core compiler distribution
  GCC 2.95.2 C++ distribution
- GDB debugger:
  ftp://sources.redhat.com/pub/gdb/releases/insight-5.2.tar.gz

Notice that the link to insight-5.0 in the ecos page is not more valid.

At that point you are ready to build the toolchain. I reported below for convenience the instructions that you can find in the ecos web page.

2.b.1 Unpacking the tools
Once the tools sources have been downloaded, they must be prepared before building. These instructions assume that the tool sources will be extracted in the `/src` directory hierarchy. Other locations may be substituted throughout. Similarly placeholders of the form *`YYYYMMDD`* and *`YYMMDD`* should be replaced with the actual date of the downloaded files. Ensure that the file system used has sufficient free space available. The contents of each archive will expand to occupy approximately 6 times the space required by the compressed archive itself. The following steps should be followed at the Cygwin *bash* prompt:

1. Create a directory for each set of tool sources, avoiding directory names containing spaces as these can confuse the build system:

```
mkdir -p /src/binutils /src/gcc /src/gdb
```

2. Extract the sources for each tool directory in turn. For *bzip2* archives:

```
cd /src/binutils
bunzip2 < binutils-2.10.1.tar.bz2 | tar xvf -
cd /src/gcc
bunzip2 < gcc-core-2.95.2.tar.bz2 | tar xvf -
bunzip2 < gcc-g++-2.95.2.tar.bz2 | tar xvf -
cd /src/gdb
bunzip2 < insight-5.2.tar.bz2 | tar xvf -
```

   The following directories should be generated and populated during the extraction process:

```
/src/binutils/binutils-2.10.1
/src/gcc/gcc-2.95.2
/src/gdb/insight-5.2
```

3. You may now need to apply a small number of source patches that are required to fix outstanding problems and add eCos support to the tools. Patches may be downloaded using most browsers by either shift-clicking or right-clicking on the link to the patch. You must not view the link and cut-and-paste because white space must be preserved exactly.

   Cygwin users should verify that their **`/tmp`** directory (or the directory specified by the *`TMPDIR`*, **`TMP`** or **`TEMP`** environment variables) is mounted in binary mode. If this is not the case the patches will apply, but the build will later fail. You may verify this using the Cygwin `mount` command. If **/tmp** is not explicitly listed, the entry for / will be used. If it says **`textmode`** for this entry, use the following command from a Cygwin bash prompt:

```
mount -f -b c:/cygwin/tmp /tmp
```

   You may need to substitute another path for **`c:/cygwin`** if Cygwin was installed to another directory.

2.b.2 Building the tools

Before attempting to build the tools, ensure that the GNU native compiler tools directory is on the PATH and precedes the current directory. The following build procedures will fail if . is ahead of the native tools in the PATH.

Avoid using spaces in build and install directory paths. Building on an NTFS file system is strongly recommended due to the large number of small files involved in the build process. Cygwin users must set the `MAKE_MODE` environment variable as follows:

```
export MAKE_MODE=UNIX
```

Approximate disk space requirements for building the development tools are as follows:

Tool	Build	Install
Binary Utilities	35MB	25MB
GCC	40MB	20MB
Insight	110MB	35MB
TOTAL	180MB	75MB

Following successful building and installation of each set of tools, the associated build tree may be deleted to save space if necessary. These instructions assume that the tools will be built in the **`/tmp/build`** directory hierarchy and installed to `/tools`. Other locations may be substituted throughout:

1. Configure the GNU Binary Utilities:

```
mkdir -p /tmp/build/binutils
cd /tmp/build/binutils
/src/binutils/binutils-2.10.1/configure --target=arm-elf \
  --prefix=/tools \
  --exec-prefix=/tools/H-i686-pc-cygwin \
  -v 2>&1 | tee configure.out
```

   If there are any problems configuring the tools, you can refer to the file `configure.out` as a permanent record of what happened.

2. Build and install the GNU Binary Utilities:

```
make -w all install 2>&1 | tee make.out
```

   If there are any problems building the tools, you can use the file **`make.out`** as a permanent record of what happened.

3. Configure GCC, ensuring that the GNU Binary Utilities are at the head of the PATH:

```
PATH=/tools/H-i686-pc-cygwin/bin:$PATH ; export PATH
```

   (for *sh*, *ksh* and *bash* users);

```
set path = ( /tools/H-i686-pc-cygwin/bin $path )
```

   (for *csh* and *tcsh* users).

```
mkdir -p /tmp/build/gcc
cd /tmp/build/gcc
/src/gcc/gcc-2.95.2/configure --target=arm-elf \
  --prefix=/tools \
  --exec-prefix=/tools/H-i686-pc-cygwin \
  --with-gnu-as --with-gnu-ld --with-newlib \
  -v 2>&1 | tee configure.out
```

4. Build and install GCC:

```
make -w all-gcc install-gcc \
  LANGUAGES="c c++" 2>&1 | tee make.out
```

5. Configure Insight:

```
mkdir -p /tmp/build/gdb
cd /tmp/build/gdb
/src/gdb/insight-5.2/configure --target=arm-elf \
  --prefix=/tools \
  --exec-prefix=/tools/H-i686-pc-cygwin \
  -v 2>&1 | tee configure.out
```

6. Build and install Insight:

```
make -w all install CC='gcc -mwin32' 2>&1 | tee make.out
```

On completion, the ARM development tool executable files will be located at **`/tools/H-i686-pc-cygwin/bin`**. This directory should be added to the head of your PATH. You can do this by editing the file **.profile** located in your home directory /home/username. The next time a cygwin shell will be started you will see the GNU executable from any directory.

2.c Installing the Apple Airport configuration software

In order to configure the access point to communicate with the SmartBadgeIV you need to configure it to be seen by the host. For this reason, if you have an additional ethernet adapter, use adapter to set – up a local network between the host and the access point by giving to the host a fixed IP address, for example 15.9.74.184.
Now you need to configure the access point so that the host can see it and give an IP address. You need the Java Run-time Environment to run the .jar file that configures the airport access point. If you don't have a JRE installed, you can download it at: http://java.sun.com/Download6 .

If you don't have the airport configuration file, you can find one at: http://edge.mcs.drexel.edu/GICL/people/sevy/airport/#Configurator .

By running the .jar file you can access to the airport configuration. Give it an IP address, for example: 15.9.74.183.

3. Target software setup

3.a Unpacking and configuring the LP-eCos distribution.

3.a.1 Packages description

LP-eCos distribution comes as a compressed .tgz archive containing the tools for configuring eCos packages and the package repository. The archive comprehends the whole package repository (version 2.0 alpha) plus the following packages targeted to the Smartbadge4:

- **CYGPKG_HAL_ARM_SA11X0_BADGE4**
  Location: **hal/arm/sa11x0/badg4/**
  Notes: Basic package to build SmartBadgeIV target.
- **CYGPKG_SA1111_BADGE4**
  Location: **devs/sa1111/**
  Notes: Package containing initialization code for the SA1111 companion chip on the SmartBadge4. This package is mandatory since the sa1111 initialization is made partially by the gdb-stub (or redboot) and must be completed by ecos.
- **CYGPKG_DEVS_PCMCIA_BADGE4**
  Location: **devs/pcmcia/arm/badge4/**
  Notes: Implements the lower layer of the pcmcia ecos driver. It is needed to support the Lucent Wavelan card package. Augmented with low power features to safely shut-down/wake-up the wavelan at run time.
- **CYGPKG_DEVS_ETH_WVLAN**
  Location: **devs/eth/wvlan/**
  Notes: Hardware dependent Lucent/Orinoco wireless lan driver. This package requires pcmcia and net packages to work. Augmented with low power features.
- **CYGPKG_AUDIO_BADGE4**
  Location: **devs/audio_tmp/**
  Notes: This package is the first version of the audio driver. A recent version can be found in the audio directory but it is under revision.
- **CYGPKG_POWMAN_BADGE4**
  Location: **/devs/powman/**
  Notes: Power Manager.

3.a.2 Unpacking and setting – up the environment

Application program under eCos can be built in two different ways: the first is intended to run with a ROM monitor. For this reason, the user application will reside in RAM. The second possibility is to build independent applications. In this case the executable will be downloaded in ROM.
In this example we take the first approach, that allows to easily test and debug user code. For this reason, we need firstly to build a rom monitor that will be downloaded in to the flash memory of the SmartBadgeIV through the JTAG port.

In order to do this, you need to have the tap utilities somewhere in your host. At least you need: **unl**, **erasefl**, **loadfl** (this version of ecos does not uses the 2nd flash bank). Now add the directory containing these utilities to your path, for example:

```
PATH=/home/username/taputils/:$PATH; export PATH
```

To get this setting permanent, modify the path line also in /home/username/.profile.

At this point you must handle a WindowsNT trick. It does not allows you to directly access I/O peripherals using user code, so you must install and run two drivers that allow to go around this problem. You can find a .zip file containing this drivers at: http://www.ddj.com/ftp/1996/1996.05/directio.zip . Once unpacked, you must run the loaddrv.exe executable and insert at the command line the full path to the file giveio.sys, which is **GIVEIO\I386\FREE\giveio.sys**. Press **install** first and **start** next.
Repeat the same procedure for totalio.sys. At that point you are able to access the JTAG port of the SmartbadgeIV through the tapmaster.

NOTE: These drivers are also mandatory to allow an user made client-server application to access the network ports.

Now you must unpack the LP-eCos distribution. To do this, open a cygwin shell and use the tar utility to extract the files for example in **/c/ecoscvs/**:

```
tar -xzf lp-ecos.tgz
```

In order to see the configuration tools from any directory under the Cygnus bask shell, modify the path variable:

```
PATH=/c/ecoscvs/ecos/:$PATH; export PATH
```

Next, define add the package repository environment variable to specify where the configuration tools find the ecos packages:

```
ECOS_REPOSITORY=/c/ecoscvs/ecos/packages/; export ECOS_REPOSITORY
```

To get permanent this setting, add the previous lines to the **.profile** file in your /home/username directory.

3.b Building and downloading the target software

3.b.1 Building and downloading Redboot

Redboot is the new rom monitor that substituted the older gdb. In order to build it, first run a bush shell and create a working directory, usually in the same directory where the ecos packages reside, for example:

```
mkdir /c/ecoscvs/ecos/ecos-stubs
```

Now cd to this directory and run the ecosconfig tool with the following parameters:

```
ecosconfig new badge4 redboot
```

After displaying some messages, an ecos.ecc file will be created. This file is a configuration script that determine what and how the ecos packages will be compiled to generate a binary redboot image. Now type:

```
ecosconfig add CYGPKG_IO_SERIAL
```

to enable the use if the serial port debugging with redboot. Then change the startup mode from RAM (default) to ROM by editing the ecos.ecc file and modifying the following line:

```
cdl_component CYG_HAL_STARTUP {
      ...
      user value ROM                          ← ←
      ...
}
```

Then perform a check of the configuration, create the build tree and make:

```
ecosconfig check
ecosconfig tree
make
```

At the end of this process, you will find a binary redboot image **redboot.bin** in **install/bin/.**

Now you are ready to download redboot. First check that your JTAG cable is properly connected to the badge4 and that the board is keep in reset state. Then power up and give the following commands:

```
unl
erasefl
loadfl  -b install/bin/redboot.bin -a0x0
```

NOTE: If you have badgelinux installed, you need also to erase the upper bank of the flash, so use unl2 and erasefl2 also.

3.b.2 Building the eCos library file and the demo application

In order to create an ecos executable, you need to create a library file that will be statically linked to the user program. To do this, create a new working directory at your bash prompt (different from the redboot one), for example:

```
mkdir /c/ecoscvs/ecos/ecos-work/
```

Now cd to this directory and give the following commands:

```
ecosconfig new badge4 net
ecosconfig import /c/ecoscvs/ecos/network.ecm
```

You are ready to build the ecos library file properly configured to build the demo application. If you need to change the networking options (ip address, netmask, ecc…), edit the **network.ecm** file now and import it again.
Now create the build tree and start building:

```
ecosconfig tree
make
```

Now you can build the demo application that can be found in the example subdirectory **/c/ecoscvs/ecos/example/MPEGnet/**. Go in this subdirectory and type:

```
make mpeg
```

3.b.3 Downloading and running with GDB

Your application is ready. Now you need to download it into the SmartBadge4 RAM. This step is accomplished by gdb, the GNU debugger. At the bash prompt give the following command:

```
arm-elf-gdb -nw /c/ecoscvs/ecos/mpeg
```

this will start the gdb debugger with a command line interface. If you omit the –nw options you will get the nicer graphical interface, but it can be not so stable depending on the host system. However, at the gdb prompt type:

```
set remotebaud 115200
target remote com1
```

Check that the SmartBadgeIV serial port is connected to the right serial port. If everything it's ok you will see an output like that:

```
Remote debugging using com1
0x400016b4 in ?? ()
```

If this is the case, type:

```
load
```

You must see the following output:

```
Loading section .rom_vectors, size 0x40, lma 20000
Loading section .text, size 0x45498, lma 0x20040
Loading section .rodata, size 0x18494, lma 654d8
Loading section .data, size 0xe11c, lma 0x7d96c
Start address 0x20040, load size 440968
Transfer rate 82040 bits/sec, 305 bytes/write.
```

Now you are ready to start the demo, but you first have to run the server program. Open a command window and go in **c:\ecoscvs\ecos\examples\Host\WINMPEGserver**. Here start the server executable:

```
feedclient.exe 9879 mp3file.mp3
```

Where mp3file is one of the available mpeg3 file you find in this directory. While the server is waiting for connections, start the demo application by typing at the gdb prompt;

```
continue
```

Measurements can be performed by connecting the DAQ probes across resistors in series with CPU, WAVELAN and system power supplies. The resulting waveforms are displayed automatically on the DAQ GUI interface.