WOSP

Wireless Open Source Platform

Andreas Karlsson

This thesis is presented as part of the Degree of Master of Science in Electrical Engineering, with emphasis on Telecommunications/Signal Processing

Blekinge Institute of Technology

December 2000

MEE 0106
ABSTRACT

The hot spot in the market of embedded systems today are opening up the systems so the world around can communicate with it. The idea of this project is to show a way to wirelessly open up an embedded system to the world. This project is both a pre-study to another project and an evaluation of Bluetooth and WAP open source components. The goal of the project is to build a prototype platform for machine-to-machine applications. This project clarifies the practical issues that rise when the idea is implemented.

To reduce the costs of the products developed on the platform and the costs for possible mass productions the platform should be based on open source software. Even the tools used should be free. Then there will be no additional license costs when the product is delivered to the ordering company.

In this project an embedded WAP-server has been developed. The server runs on top of Axis’ Bluetooth communication stack. The practical aspects of WAP over Bluetooth are a field in which not much experience is present. This project among one in Ericsson are the only one known in the world today that runs WAP over Bluetooth.

All mobile phone and PDA users having access to a non-cost browsing tool, that WAP over Bluetooth is, would certainly strengthen both WAP and Bluetooth and perhaps even stake out the path to a new market. The market of devices with embedded systems that would benefit from being opened up to the world with the possibility of running WAP directly over Bluetooth couldn’t be negligible. Examples of use cases are wireless wallets, remote controls and when a serviceman should fix a machine that is out of order or just performs some services.

The operating system used is eCos, which is open source and shipped by Red Hat. The system runs on an evaluation board from ARM equipped with an ARM7 processor, 256 KB RAM and 256 KB ROM.

The result of this project is a demo system that with helps of a WAP-browser makes it possible to retrieve information from an embedded system and send commands back to it.
# CONTENTS

ABSTRACT .................................................................................. 1  
CONTENTS .................................................................................. 3  
1 INTRODUCTION ........................................................................ 5  
  1.1 PLATFORM REQUIREMENTS ................................................... 6  
  1.2 OPEN SOURCE ....................................................................... 7  
     1.2.1 Open Source Successful Products ......................................... 8  
     1.2.2 Open Source Business Models ................................................ 8  
2 PLATFORM COMPONENTS ......................................................... 9  
  2.1 MICROCONTROLLER ............................................................. 9  
  2.2 OPERATING SYSTEM ............................................................ 9  
     2.2.1 Four Paths to Real-Time Linux ............................................... 10  
     2.2.2 eCos ................................................................................. 11  
     2.2.3 Cygwin .............................................................................. 15  
3 PROTOCOL OVERVIEWS ........................................................... 17  
  3.1 BLUETOOTH STACK OVERVIEW ........................................... 17  
     3.1.1 Baseband (BB) .................................................................. 17  
     3.1.2 Link Manager Protocol (LMP) ............................................... 18  
     3.1.3 Host Control Interface (HCI) ............................................... 18  
     3.1.4 Logical Link Control and Adaptation Protocol (L2CAP) .......... 18  
     3.1.5 Service Discovery Protocol (SDP) ......................................... 18  
     3.1.6 Cable Replacement Protocol (RFCOMM) ............................. 19  
     3.1.7 Adopted Protocols ............................................................. 19  
  3.2 TCP/IP STACK OVERVIEW .................................................... 19  
     3.2.1 Point-to-Point Protocol (PPP) ............................................... 20  
     3.2.2 Internet Protocol (IP) .......................................................... 21  
     3.2.3 Transmission Control Protocol (TCP) .................................. 21  
     3.2.4 User Datagram Protocol (UDP) ......................................... 22  
  3.3 WAP STACK OVERVIEW ....................................................... 23  
     3.3.1 Wireless Datagram Protocol (WDP) ..................................... 24  
     3.3.2 Wireless Transport Layer Security (WTLS) .......................... 24  
     3.3.3 Wireless Transaction Protocol (WTP) .................................. 24  
     3.3.4 Wireless Session Protocol (WSP) ......................................... 25  
     3.3.5 Wireless Application Environment (WAE) ............................ 25  
  3.4 HOW WAP IS INTENDED TO WORK ABOVE BLUETOOTH ........ 25  
4 PORTING TO WOSP ................................................................. 27  
  4.1 PORTING AXIS’ BLUETOOTH STACK ................................... 27  
     4.1.1 Short Description of the Stack .............................................. 27  
     4.1.2 Modifications to Axis’ Stack ............................................... 29  
  4.2 PORTING UDP/IP/PPP STACK .............................................. 32  
  4.3 MEMORY EXPANSION .......................................................... 32  
  4.4 PORTING THE WAP STACK .................................................. 33  
     4.4.1 Short description of Ophelia .................................................. 34  
     4.4.1.1 Architecture of Ophelia ...................................................... 34  
     4.4.1.2 Operational Procedures .................................................... 35  
     4.4.1.3 Layer Communication ...................................................... 35  
     4.4.2 Modifications to Ophelia’s Stack ......................................... 36  

---

WOSP-Wireless Open Source Platform
5 REDESIGN OF WOSP ................................................................. 39
  5.1 REASONS TO THE REDESIGN ............................................. 39
  5.2 CONCLUSIONS OF THE REDESIGN ...................................... 40
  5.3 WOSP’S DEMO SYSTEM ......................................................... 41
      5.3.1 System overview ........................................................... 43
6 OUTLOOK ................................................................................... 47
7 REFERENCES ................................................................................. 49
8 APPENDIX A ................................................................................. 51
   8.1 QUESTION AND ANSWERS ABOUT eCos ................................. 51
9 APPENDIX B ................................................................................. 55
   9.1 INSTALLING GNU TOOLKIT .................................................. 55
       Problem 1: .............................................................................. 55
       Solution: .................................................................................. 55
       Problem 2: .............................................................................. 56
       Solution: .................................................................................. 56
   9.2 GETTING eCos RUNNING ....................................................... 56
       Alternative 1 – prompt ............................................................ 57
       Alternative 2 – graphical .......................................................... 57
   9.3 DOWNLOADING THE PROGRAM WITHOUT GDB .................... 58
       9.3.1 Configuring eCos .............................................................. 58
       9.3.2 Build your sources ............................................................ 58
       9.3.3 Edit the preferences in the windows based uuencoder ............ 60
       9.3.4 Download the uuencoded file ............................................ 60
10 APPENDIX C ................................................................................. 61
   10.1 MODIFICATIONS TO COMPIL eCos’ BLUETOOTH STACK ......... 61
   10.2 MODIFICATIONS TO eCos SOURCES .................................... 62
11 APPENDIX D ................................................................................. 63
   11.1 EXPANDING AEB-1’S MEMORY ............................................ 63
   11.2 USING eCos WITH THE NEW MEMORY .................................. 64
   11.3 BUILDING THE STUB UNDER WINDOWS ............................... 66
   11.4 USING THE NEW STUB ......................................................... 66
12 APPENDIX E ................................................................................. 67
   12.1 INTERRUPT MODEL IN eCos ................................................ 67
13 APPENDIX F ................................................................................. 69
   13.1 MODIFICATIONS TO COMPIL eCos THE WAP STACK ............ 69
1 INTRODUCTION

The term "embedded technology" generally conjures up images of one manufacturer's microprocessors, chipsets, or boards integrated into another manufacturer's product: a cellular phone, pager, electronic game or industrial devices such as a telecommunications switch or a multifunction copier. While such single-task products continue to make up the biggest share of the embedded market, the market is experiencing tremendous growth and innovation, with no end in sight.

More and more products employ embedded technology every day, we are now seeing a number of innovative new products unimaginable just a few years ago. Like Web phones, for example: intelligent telephones with a touch screen and small keyboard that offer Web browsing and e-mail besides voice communications.

In many products containing embedded systems their functionality may be such that their presence is far from obvious to the casual observer.

But it is not just the embedded market that is growing; the technology itself has evolved – and continues to evolve – to become far more network-centric. Traditionally embedded technology tended to have limited functionality, often designed to perform a single task. However, today's embedded customers demand more flexible real-time solutions that are highly scalable, extremely reliable, and most importantly, network-ready.

The hot spot in the market of embedded systems today are opening up the systems so the world around can communicate with it. Why not make a phone call to your coffee maker at home to double check that it is turned off?

It becomes even hotter when the device is connected to the world wirelessly. Why not make a phone call to your car to see if it is locked or check the fuel level?

We have only scratched the surface of the enormous possibilities of the new and emerging technologies.

The embedded system market is a fast growing market where the competition is growing harder and harder. To succeed on this market you do not only need a very good unique idea. You also need to launch your product before anyone else does, which often means a schedule under extreme pressure. Even with hard requirement on the time to market the product still must be reliable. Since the development division in a company often is undermanned or/and not has the demanded experience in the field of embedded systems consultants are needed. Combitech Systems AB is a consultant company offering high competence in developing embedded real-time systems, based on extensive experience and in-depth knowledge.
Every sign on the market points in one direction, Bluetooth will become very popular. Combitech Systems AB plays a leading role on this new highly interesting field when companies want to integrate Bluetooth into their products. It is not an easy task to predict which ideas Combitech Systems AB will be asked to realize. A platform, serving as a base for new ideas would reduce time to market.

This outline of this report will be as follows. First a short description of the requirements and the environment will be given. Then a very brief introduction to all the protocols used. Finally the main task of this project, the porting and integration of several software components will be described. Details can be found in appendix as will be pointed out in the text.

1.1 PLATFORM REQUIREMENTS

To reduce the costs of the products developed on the platform and the costs for possible mass productions the platform should be based on open source software. Even the tools used should be free. Then there will be no additional license costs when the product is delivered to the ordering company.

The platform should be based on a well-supported micro controller and a small-memory operating system. Additional requirement on the operating system is that it should be ported to other microprocessors than ARM, have small memory needs and if possible include a TCP/IP stack.

On top of this base will a Bluetooth stack be putted and above Bluetooth will UDP/IP be located to follow the LAN-profile. The goal is to be able to browse into the embedded system on a small wireless device to check some interesting parameters and therefore WAP will be placed above the other protocols.

Many embedded systems do not have any operating system at all, just a control loop. This may be sufficient for very simple ones; however, as systems grow in complexity, an operating system becomes essential or the software grows unreasonably complex. Since this will be a not product specific platform an operating system will make development easier. The fact that both Bluetooth and TCP/IP are used makes the use of an operating system even more justified.

Software components:
- Development Tools
  - Cygwin.
  - GNU Compiler.
  - GNU Debugger.
- Operating System
  - Ecos
- Axis’ Bluetooth stack
  - SDP, Service Discovery Protocol.
  - RFCOMM, Cable Replacement Protocol.
  - L2CAP, Logic Link Control and Adaptation Protocol.
WOSP-Wireless Open Source Platform

- HCI, Host Controller Interface.
- TCP/IP stack
  - The stack shipped with eCos should be used.
- WAP stack
  - Kannel or another open source stack should be used

Hardware components:
Evaluation board equipped with ARM7

1.2 OPEN SOURCE

The term “Open Source” is one of the most used out on the Internet these days. The idea of open source can be hard to understand by old-fashion business people. Their opinion is- “A software company develops only software, if this software is delivered for free to everybody who wants it, how can the company earn any money?”

One answer to this question is: “Open Source” is a way that many companies and individuals can collaborate on a product that none of them could achieve alone. In this way the product will get a faster time to market and be way more reliable.

There is an organisation behind the term, http://www.opensource.org. On this page they state “The intent of the Open Source Definition is to write down a concrete set of criteria that we believe capture the essence of what the software development community wants “Open Source” to mean -- criteria that ensure that software distributed under an open-source license will be available for independent peer review and continuous evolutionary improvement and selection, reaching levels of reliability and power no closed product can attain.”

The basic idea behind open source is very simple. When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. When a lot of persons run the code, the software will be under rigorous tests since not everyone uses the product in the same way.

An important side effect of the open-source model will be a much wider platform range for your product. Open-source authors frequently find themselves receiving, for free, port changes for operating systems and environments they barely know exist and can’t afford developers to support. Each such port, of course, widens the market appeal of the product. If it is a useful product developed on a specific operating system someone will almost certainly port it to another operating system e.g. the TCP/IP stack developed by Berkeley in the early 80’s. The same happens a useful product ported to a specific hardware a processor or whatever.

And this can happen at a cost of zero and a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

Like with everything else there are negative sides of the concept. One programmer often creates code which fits his needs and do not consider other platforms/ operating system which his changes can harm. When several programmers get write access to the source code repository it can be hard to check that the added code is compatible and without any errors. One solution
is to have a few masters, with write access to the repository, to whom all
updates are sent. This solution catches the obvious bugs before it is merged
into the repository.

1.2.1 Open Source Successful Products
The Internet's infrastructure makes the best possible refutation.
- The various open-source TCP/IP stacks and utility suites
- Apache, http://www.apache.org/, which runs over 50% of the world's
web servers.
- Perl, http://www.perl.org/, which is the engine behind most of the `live
content' on the World Wide Web.
- BIND, http://www.isc.org/products/BIND/, the software that provides
the DNS (Domain Name Service) for the entire Internet.
- sendmail, http://www.sendmail.org/, the most important and widely
used email transport software on the Internet.

1.2.2 Open Source Business Models
There are at least four known business models for making money with open
source:

1. Support Sellers
   In this model, you give away the software product, but sell distribution,
   branding, and after-sale service. This is what for example Red Hat and
   Cygnus are doing.

2. Loss Leader
   In this model, you give away open-source as a loss leader and market
   positioner for closed software. This is what Netscape is doing.

3. Widget Frosting
   In this model, a hardware company (for which software is a necessary
   adjunct but strictly a cost rather than profit center) goes open-source in
   order to get better drivers and interface tools cheaper. Silicon Graphics, for
   example, supports and ships Samba.

4. Accessorizing
   Selling accessories -- books, compatible hardware, complete systems with
   open-source software pre-installed. It is easy to trivialize this (open-source
   T-shirts, coffee mugs, Linux penguin dolls) but at least the books and
   hardware underlay some clear successes: O'Reilly Associates, SSC, and
   VA Research are among them.

The open-source culture's exemplars of commercial success have, so far, been
support sellers or loss leaders. Nevertheless, some people think there is good
reasons to believe that the clearest near-term gains in open-source will be in
widget frosting. For widget-makers (such as semiconductor or peripheral-card
manufacturers), interface software is not even potentially a revenue source.
Therefore the downside of moving to open source is minimal.
2 PLATFORM COMPONENTS

The range of software being open source varies heavily from one component to another. For example there are not that many free Bluetooth stacks, only Axis Communications’ stack was found during the research. On the other hand there seems to be no limit on how many free versions of embedded Linux operating systems there exist. In this section the two fundamental choices of processor and operating system specified in the requirements will be justified and given an introduction.

2.1 MICROCONTROLLER

ARM's microprocessor cores have rapidly become the volume RISC standard in such markets as portable communications, hand-held computing, multi-media digital consumer and embedded solutions. ARM is a Bluetooth partner to Ericsson, Ericsson has for example, licensed the ARM7TDMI® processor core for use in its communication products, http://www.arm.com/. Since Ericsson is a significant member of the Bluetooth Special Interest Group (SIG), they invented Bluetooth, and a considerable customer to Combitech Systems AB ARM's microprocessors was preferred in this project. Besides ARM architecture is well supported by semiconductor partners, real-time operating system providers, third-party toolchain developers and application software providers.

The ARM evaluation board (AEB-1 revC) was chosen. This board includes:

- 256K Flash EEPROM
- 256K SRAM

For more information see http://www.arm.com/Documentation/Overviews/AEB/.

2.2 OPERATING SYSTEM

There are a few dozen viable commercial operating systems from which to choose. A few big players have emerged, such as VxWorks, pSOS, Neculeus , Enea OSE and Windows CE.

In addition to a variety of commercial operating systems, there are an amazing number of proprietary operating systems. Many of these are created from scratch, such as Cisco's IOS; others are derived from some other operating system. For example, many network products are derived from the same version of the Berkeley UNIX system, because it has complete networking capability.
Linux as an embedded OS is a new candidate with some attractive advantages. It is portable to many CPUs and hardware platforms, stable, scalable over a wide range of capabilities and easy to use for development.

Several operating systems have been ported to the, by industry, well received ARM7DI processor core. Some of them, besides the big players mentioned before, are RTXC, ChorusOS, IxWorks, VRTX/mc, OSE™, FLEX OS etc. A complete list can be found at http://www.arm.com/DevSupp/OS/.

Some of these operating systems listed are open source. Since Linux variants come with a great deal of support through mailing lists and have proven to adapt to new technologies early a Linux variant was preferred.

2.2.1 Four Paths to Real-Time Linux

Linux, in its present incarnation (2.2/2.4 kernel derivatives), is not a real time operating system by traditional definitions. The Linux design philosophy has emphasized throughput over responsiveness and determinism: its scheduler employs exhaustive fairness-based thread selection and can promote low-priority threads to the head of the ready list, per its own criteria, to avoid starvation.

Moreover, the Linux kernel itself is non-preemptible - no rescheduling can take place until the kernel completes a system call or other internal activity. Given this history here is four paths to use Linux as a real-time operating system.

**Linux as is:**
In soft real-time systems where Windows NT or Sun/Solaris is running Linux can be used as is. Linux excels in price/performance. Linux holds its own even with as little as 8-16 MB RAM and sub-200 MHz clock speeds.

**Giving Linux a Tune-Up: Device Driver Optimization**
In many, if not all, applications with definable real-time requirement, this requirement is constrained to definable portions of the hardware and software environment such as handling audio or video packets in a streaming application. The Linux kernel and device drivers can be tuned to offer needed responsiveness. Still the requirements of a hard real-time system can’t be fulfilled.

**Kernel Substitution**
Inserting a second kernel, as occurs with the use of RTLinux and Real-time Application Interface (RTAI), is merely a more generalized approach to device driver tuning. Instead of just tweaking a few critical hardware interfaces, these real-time add-ons provide a generalized RTOS API and threading mode, while relegating the main Linux kernel to the status of a low-priority thread under their control. This is shown in figure 2.1.
RTLinux, RTAI and comparable strategies are best employed when the mainstream Linux APIs are leveraged for only non-time-critical housekeeping operations. The two do offer excellent nominal latencies with reported performance in the 30-50-microsecond range. Because this architecture runs a standard Linux kernel as a low-priority real-time task, any Linux application can be run in this environment while communicating with hard real-time tasks that are the focus of the system. The memory size of this architecture is not suitable to most embedded systems.

**Scheduler Enhancement: Preserving Linux APIs**

The cleanest path to real-time is one that neither breaks Linux APIs, nor introduces one more set of APIs into the picture. Thus, the ideal means to meeting real-time requirements is to make Linux itself more responsive.

A Monta Vista Software project enhances the scheduler. The Monta Vista schedule runs in front of the default-exhaustive fairness-based scheduler, enforcing hard prioritization and policies on all calls to the Linux scheduler. Should no identifiable real-time threads be available for execution, then control is passed to the standard scheduler.

This API transparent scheme does not address the issue of kernel preemption. As a result it only offers soft real-time responsiveness: sub millisecond timings with context switching in the 100-150-microsecond range.

Linux has a distance to go before it can be declared ready to take on the challenges of hard real-time applications with requirements on a small memory footprint.

**2.2.2 eCos**

Since Linux currently scales from a minimal size of around 500 kilobytes of kernel and 1.5MB of RAM, all before taking into consideration application and service requirements. Red Hat has developed an open source real-time operating system for deeply embedded applications. It meets the requirements of the embedded space that Linux cannot yet reach. ECos is a streamlined operating system supported by the Red Hat GNUPro and GNU open source
development tools. This is a highly configurable operating system, eCos provides source-level configuration to exactly match application needs, which means that you can include exactly the parts of the operating system you actually need. This configuration is done via a graphical interface in Windows as shown in figure 2.2 and 2.3. When using eCos on Linux there is no graphical interface yet instead one has to edit a text file. The fewer the requirements are the smaller the kernel will be. A typical kernel that includes a scheduler, memory management, RTC support, several threads, and interrupt handlers would require about 3K ROM and 1K RAM, incredible!

![Figure 2.2: eCos Package Tool](image-url)
ECoS does not have a deadline based scheduler with guarantees, so it is not a hard real time OS. It does have a multi priority level scheduler. If you make your hard real time tasks run with higher priority than your soft real time threads it will do its best to run those first until they block. If you have too many hard real-time threads it will run out of CPU.

**EL/IX related functionality**

EL/IX (Embedded Linux/ PosIX) is an API that allows applications to be ported transparently between Linux, Embedded Linux and small embedded OS's like eCos. Visit [http://sources.redhat.com/elix](http://sources.redhat.com/elix) for details.

- Ecos supports POSIX 1003.1 threads, mutexes, semaphores, message queues, condition variables, clocks, timers and real-time signals in the new POSIX compatibility package (part of the EL/IX initiative).
- Plug-in filesystem support, with a POSIX 1003.1 compatible API, found in the io/fileio package, and integrated with the C library.
- An initial example filesystem implementation is provided in fs/ram. This is a RAM-based filesystem that attempts to both be a useful tool in its own right and an example of how to write a filesystem that plugs into the eCos filesystem infrastructure.
- POSIX termios (tty controller) support as specified by EL/IX level 1.
Networking functionality
The TCP/IP stack is derived from the OpenBSD source base and provides UDP, TCP, ICMP and BOOTP protocol support on an IPv4 standards base. Device driver support for Cirrus Logic EP72xx evaluation boards and Motorola MBX is included. The stack, ethernet core support, and device drivers are all distributed as configurable eCos packages.

A port of the University of California at Davis (UCD) SNMP is included. DHCP and TFTP server and client support is now included. The GoAhead web server has been ported to eCos http://www.goahead.com/.

RedBoot
RedBoot - the Red Hat Embedded Debug and Boot ROM monitor has been added to eCos.
RedBoot provides:
• Serial and ethernet GDB debugging support
• Flash image management
• Flash booting
• Download via TFTP
• SREC or "raw" download formats
• IP address setting by BOOTP/DHCP
• A Command Line Interface (CLI).

Zmodem download support is to be added shortly. Zmodem is a way to transfer information between PCs and embedded computers, http://www.omen.com/ind.html.
RedBoot will become the standard firmware supplied and supported by Red Hat, replacing CygMon and GDB stubs based boot loaders.
Initial support is for the ARM-EDB7xxx, ARM-EBSA285, and StrongARM-Assabet boards.

ECos in WOSP
In WOSP eCos v 1.3.1 ran in RAM with GDB support and one serial driver enabled, the approximated downloaded size of eCos compiled using –O2 is 28 KB. The operating system worked very well during the project. One thing with eCos that comes from the feature of being Open Source is that the support is tremendous even without paying for a support-contract with Red Hat. The WOSP project has really had benefit from the eCos mailing list, which is very active.
The documentation is good. All the latest documentation can be found on the web. Other people’s experiences of eCos can be read in Appendix A.
Information about eCos can be found at the addresses http://sources.redhat.com/ecos/ and http://www.redhat.com/products/ecos/.

Examples of other supported targets are Fujitsu SPARCLite 32-bit product family, Hitachi 32-bit SuperH RISC processors, i386 architecture processors, Matsushita N10300 32-bit microcontroller series, NEC V800 Series 32-bit microcontrollers, NEC VR Series 64-bit MIPS RISC microprocessors, Toshiba TX39 32-bit TX SystemRISC series and PowerPC family. See
http://sourceware.cygnus.com/ecos/hardware.html for the very latest list of supported architectures.

2.2.3 Cygwin
To install the GNU tools required by eCos then Cygwin must be installed first. The Cygwin tools are ports of the popular GNU development tools and utilities for Windows 95, 98, and NT. They function by using the Cygwin library, which provides an UNIX-like API on top of the Win32 API [Cygwin]. As a result, it is possible to easily port many significant Unix programs without the need for extensive changes to the source code. This includes configuring and building most of the available GNU software (including the packages included with the Cygwin development tools themselves). Many standard Unix utilities are provided with the package for example, ls, rm, mv, tar, bzip2, less etc. They can be used both from the bash shell and from the standard Windows command shell. See http://sources.redhat.com/cygwin/. Information gathered during the WOSP project on how to install and getting started with eCos and Cygwin can be found in Appendix B.
3 PROTOCOL OVERVIEWS

3.1 BLUETOOTH STACK OVERVIEW

As mentioned before the stack from Axis Communications will be used as a base. Unfortunately this stack is only ported to Linux operating system and not eCos so a special porting has to be done. This porting procedure is described in next section. In this section a short description will be given of each layer of Bluetooth in the LAN-profile.

During the requirement specification phase of WOSP the LAN access profile was chosen as the natural profile to be used. Since this project is not intended as a Bluetooth description the layers in the Bluetooth-stack will be given a very short explanation as done in the white paper [Riku99]. Figure 3.1 shows the mutual relationship between the different protocols.

![Figure 3.1: Protocol Overview of the LAN-Profile](image)

3.1.1 Baseband (BB)

The Baseband and Link Control layer enables the physical RF link between Bluetooth units forming a piconet [Bluetooth]. As the Bluetooth RF system is a Frequency-Hopping-Spread-Spectrum system in which packets are transmitted in defined time slots on defined frequencies, this layer uses inquiry and paging procedures to synchronize the transmission hopping frequency and clock of different Bluetooth devices.

The reason to choose a Frequency-Hopping-Spread-Spectrum is that Bluetooth operates in the globally license-free Industrial-Scientific-Medical (ISM) band at 2.45 GHz. Since this band is open to anyone, radio systems operating in this band must cope with several unpredictable sources of interference, such as baby monitors, garage door openers, cordless phones and microwave ovens (the strongest source of interference).

The frequency hopping uses a high hopping rate of 1600 hops/s. If a packet is lost, only a small portion of the message is lost. Packets can also be protected...
by forward error control (FEC). Voice is never retransmitted. Instead a robust voice-encoding scheme that is based on continuous variable slope delta (CVSD) modulation is used. This layer provides two different kinds of physical links with their corresponding Baseband packets, Synchronous Connection-Oriented (SCO) and Asynchronous Connectionless (ACL), which can be transmitted in a multiplexing manner on the same RF link. ACL packets are used for data only, while the SCO packet can contain audio only or a combination of audio and data. All audio and data packets can be provided with different levels of FEC or CRC error correction and can be encrypted. Furthermore, the different data types, including link management and control messages, are each allocated a special channel.

3.1.2 Link Manager Protocol (LMP)
The link manager protocol [Bluetooth] is responsible for link set-up between Bluetooth devices. This includes security aspects like authentication and encryption by generating, exchanging and checking of link and encryption keys and the control and negotiation of baseband packet sizes. Furthermore it controls the power modes and duty cycles of the Bluetooth radio device, and the connection states of a Bluetooth unit in a piconet.

3.1.3 Host Control Interface (HCI)
HCI provides a command interface to the baseband controller, link manager, and access to hardware status and control registers. Often HCI is positioned below L2CAP but this positioning is not mandatory but HCI can exist e.g., above L2CAP.

3.1.4 Logical Link Control and Adaptation Protocol (L2CAP)
The Bluetooth logical link control and adaptation protocol (L2CAP) [Bluetooth] adapts upper layer protocols over the baseband. It can be thought to work in parallel with LMP in difference that L2CAP provides services to the upper layer when the payload data is never sent at LMP messages. L2CAP provides connection-oriented and connectionless data services to the upper layer protocols with protocol multiplexing capability, segmentation and reassembly operation, and group abstractions. L2CAP permits higher-level protocols and applications to transmit and receive L2CAP data packets up to 64 kilobytes in length. Although the Baseband protocol provides the SCO and ACL link types, L2CAP is defined only for ACL links and no support for SCO links is specified in Bluetooth Specification 1.0.

3.1.5 Service Discovery Protocol (SDP)
Discovery services are crucial part of the Bluetooth framework. These services provide the basis for all the usage models. Using SDP, device information, services and the characteristics of the services can be queried and after that, a connection between two or more Bluetooth devices can be established. SDP is defined in the Service Discovery Protocol specification [Bluetooth].

3.1.6 Cable Replacement Protocol (RFCOMM)
RFCOMM is a serial line emulation protocol and is based on ETSI 07.10 specification. This “cable replacement” protocol emulates RS-232 control and data signals over Bluetooth baseband, providing both transport capabilities for upper level services (e.g. OBEX) that use serial line as transport mechanism. RFCOMM is specified in [Bluetooth].

3.1.7 Adopted Protocols
In designing the protocols and the whole protocol stack, the main principle has been to maximize the re-use of existing protocols for different purposes at the higher layers, instead of re-inventing the wheel once again. The protocol re-use also helps to adapt existing (legacy) applications to work with the Bluetooth technology and to ensure the smooth operation and interoperability of these applications. Thus, many applications already developed by vendors can take immediate advantage of hardware and software systems, which are compliant to the Specification. The Specification is also open, which makes it possible for vendors to freely implement their own (proprietary) or commonly used application protocols on the top of the Bluetooth-specific protocols. Thus, the open Specification permits the development of a large number of new applications that take full advantage of the capabilities of the Bluetooth technology. Some of the adopted protocols will be handled in next section.

3.2 TCP/IP STACK OVERVIEW
As shown in figure 3.1 the protocols UDP, IP and PPP should be added on top of Bluetooth. These protocol standards are defined by the Internet Engineering Task Force and used for communication across the Internet [IETF]. Now considered as the most widely used protocol family in the world, TCP/IP stacks have appeared on numerous devices including printers, handheld computers, and mobile handsets. Access to these protocols is operating system independent, although traditionally realized using a socket programming interface model. The implementation of these standards in Bluetooth devices allows for communication with any other device connected to the Internet: The Bluetooth device, should it be a Bluetooth cellular handset or a data access point for example is then used as a bridge to the Internet. TCP/IP/PPP is used for the all Internet Bridge usage scenarios in Bluetooth 1.0 and for OBEX in future versions [Bluetooth]. UDP/IP/PPP is also available as transport for WAP [Bluetooth]. Next section will handle the use of UDP/IP/PPP in WOSP. A short description of the most important protocols in the TCP/IP-stack will follow. Figure 3.2 gives an overview of how the TCP/IP-stack corresponds to the OSI-stack and where each layer should be placed in these layered structures.
3.2.1 Point-to-Point Protocol (PPP)

In the Bluetooth technology, PPP is designed to run over RFCOMM to accomplish point-to-point connections. PPP is the IETF Point-to-Point Protocol. PPP-Networking is the means of taking IP packets to/from the PPP layer and placing them onto the LAN. Usage of PPP over Bluetooth is described in the LAN Access Profile.

PPP is a protocol for communication between two computers using a serial interface, typically a personal computer connected by phone line to a server. PPP uses the Internet protocol (IP) (and is designed to handle others). It is sometimes considered a member of the TCP/IP suite of protocols. Relative to the Open Systems Interconnection (OSI) reference model, PPP provides layer 2 (data-link layer) services.

PPP enables to connect computers and networks at separate physical locations by using modems and telephone lines. With PPP, users with computers at home or in remote offices can connect to site's network. You can also use the combination of PPP software, a modem, and telephone lines as a router connecting networks in different places. PPP offers strategies for configuring these machines and networks. PPP is a full-duplex, bit-oriented protocol that can run over synchronous or asynchronous links. PPP uses a variant of High Speed Data Link Control (HDLC) as the basis for encapsulation. Links may be dedicated or circuit-switched, and PPP can work over copper, fiber optic, microwave, or satellite leased lines. PPP provides data error detection while higher layer protocols are responsible for error recovery.

PPP is usually preferred over the earlier de facto standard Serial Line Internet Protocol (SLIP) because it can handle synchronous as well as asynchronous communication. PPP can share a line with other users and it has error detection that SLIP lacks.
3.2.2 Internet Protocol (IP)

IP [IP] is the central, unifying protocol in the TCP/IP suite. It provides the basic delivery mechanism for packets of data sent between all systems on an internet, regardless of whether the systems are in the same room or on opposite sides of the world. All other protocols in the TCP/IP suite depend on IP to carry out the fundamental function of moving packets across the Internet.

In terms of the OSI networking model, IP provides a Connectionless Unacknowledged Network Service, which means that its attitude to data packets can be characterised as "send and forget". IP does not guarantee to actually deliver the data to the destination, nor does it guarantee that the data will be delivered undamaged, nor does it guarantee that data packets will be delivered to the destination in the order in which they were sent by the source, nor does it guarantee that only one copy of the data will be delivered to the destination.

Because it makes so few guarantees, IP is a very simple protocol. This means that it can be implemented fairly easily and can run on systems that have modest processing power and small amounts of memory. It also means that IP demands only minimal functionality from the underlying medium (the physical network that carries packets on behalf of IP) and can be deployed on a wide variety of networking technologies.

The no-promises type of service offered by IP is not directly useful to many applications. Applications usually depend on TCP or UDP to provide assurances of data integrity and (in TCP's case) ordered and complete data delivery.

The only thing that IP cares strongly about is the maximum size of a frame that can be carried on the medium. This controls whether, and to what extent, IP must break down large data packets into a train of smaller packets before arranging for them to be transmitted on the medium. This activity is called fragmentation and the resulting smaller and incomplete packets are called "fragments". The final destination is responsible for rebuilding the original IP packet from its fragments, an activity called fragment reassembly.

To avoid the time consuming parts included in fragmentation and reassembly the solution should keep the maximum sizes equivalent for RFCOMM and PPP. Further the applications should never send any data bigger than this maximum size. Also according to the fact that UDP does not keep track of the packets’ sequence numbers we should avoid fragmentation.

3.2.3 Transmission Control Protocol (TCP)

TCP [TCP] is a sophisticated, reliable, byte-stream protocol. TCP is responsible for making sure that the commands get through to the other end. It keeps track of what is sent, and retransmits anything that did not get through. If any message is too large for one datagram, TCP will split it up into several datagrams, and make sure that they all arrive correctly. One can think of TCP as forming a library of routines that applications can use when they need reliable network communications with another computer.
- TCP provides reliability. The TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted, and TCP waits a longer amount of time. After some number of retransmissions, TCP will give up, with the total amount of time spent trying to send data typically between 4 and 10 minutes (implementation depended).

- TCP provides flow control. TCP always tells its peer exactly how many bytes of data it is willing to accept from the peer. This is achieved by returning a window with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. At any time, the window is the amount of room currently available in the receive buffer, guaranteeing that the sender cannot overflow the receiver's buffer.

- TCP connection is full-duplex. This means that the application can send and receive data in both directions on a given connection at any time.

- TCP provides multiplexing and demultiplexing. To allow for many processes within a single host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket.

- TCP provides connections between clients and servers. The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection. A pair of sockets identifying the two sides uniquely specifies a connection. That is, a socket may be simultaneously used in multiple connections. When two processes wish to communicate, their TCPs must first establish a connection (initialize the status information on each side). When their communication is complete, the connection is terminated or closed to free the resources for other uses.

3.2.4 User Datagram Protocol (UDP)

Although the services that TCP supplies are needed by many applications, there are still some kinds of applications that do not need them. In our case all functionality is in the WAP protocols and UDP/IP serves as a bearer.

UDP [UDP] is a simple, unreliable, datagram protocol. The application writes a datagram to a UDP socket, which is encapsulated as an IP datagram, which is then sent to its destination. But there is no guarantee that a UDP datagram ever reaches its final destination. If an application wants to be certain that a datagram reaches its destination, lots of features must be built into it: acknowledgments from the other end, timeouts, retransmissions, and the like.
UDP allows the application to send and receive only individual packets of data. Each packet is sent as a separate transmission. UDP only provides a simple interface to the IP layer. It has no mechanism to retry or retransmit lost packets. If the application has a large amount of data to send, it has to worry about breaking up the data into individual, small-enough pieces. In the network layer, IP does allow for fragmentation and reassembly, but generally UDP applications pay attention to the maximum transmission unit. Also, the networking infrastructure in between the sending and receiving machines can change routing decisions at any time; therefore the application can make no assumption that the packets were received in the order that they were sent.

UDP provides a connectionless service, as there need not be any long-term relationship between a UDP client and server.

### 3.3 WAP STACK OVERVIEW

A short description of the WAP-protocols will be given and then an example on how WAP is intended to work over Bluetooth.

The Wireless Application Protocol (WAP) Forum is building a wireless protocol specification [WAP] that works across a variety of wide-area wireless network technologies. Bluetooth and WAP technologies are not in any way opposing each other; in fact they both will strengthen by collaboration.

The goal of WAP is to bring Internet content and telephony services to digital cellular phones and other wireless terminals. In Figure 3.3, the protocol stack of the WAP framework is depicted.

![Figure 3.3: WAP stack](image)

Bluetooth as a WAP Bearer is defined in [Bluetooth]. Hidden computing usage models can be implemented using the WAP features.

The idea behind the choice of WAP is to reuse the upper software applications developed for the WAP Application Environment (WAE). These include WML (Wireless Markup Language) and WTA (Wireless Telephony Application) browsers that can interact with applications on the PC. Building
application gateways which mediate between WAP servers and some other application on the PC makes it possible to implement various hidden computing functionality, like remote control, data fetching from PC to handset etc. WAP servers also allow for both content push and pull between PC and handset, bringing to life concepts like information kiosks. WAP framework also opens up the possibility of custom applications for handsets that use WML and WML Script as "universal" Software Development Kit.

3.3.1 Wireless Datagram Protocol (WDP)
WDP is a transport layer protocol that offers transport services, and allows WAP to be bearer independent. WDP operates above bearer services. The bearer service that is used depends on the network type. WDP provides an interface to the upper layer protocols. This makes it possible for the Security, Session, and Application layers to function independently of underlying wireless network type.

3.3.2 Wireless Transport Layer Security (WTLS)
WTLS is a security protocol. It is based on the Transport Layer Security (TLS) protocol. WTLS is optimized for use over narrow-band communication channels. WTLS is to be used with the WAP transport protocols. WTLS provides:

- **Data Integrity** - WTLS ensures that data sent between a terminal and an application server remains unchanged and uncorrupted.
- **Privacy** - WTLS ensures that data transmitted between a terminal and an application server is private. Any intermediate parties that may have intercepted the data stream cannot understand the data.
- **Authentication** - WTLS can establish the authenticity of a terminal and an application server.
- **Denial-of-service protection** - WTLS can reject and detect data that is replayed or not successfully verified. WTLS protects the upper protocol layers, and makes many typical denial-of-service attacks hard to accomplish.
- WTLS may be used for authentication of electronic business card exchange.

3.3.3 Wireless Transaction Protocol (WTP)
The WTP protocol provides a transaction-oriented protocol for implementation in mobile stations. WTP runs on top of a datagram service. WTP operates over secure or non-secure wireless datagram networks and provides:

- Three different classes of transaction services: Unreliable and reliable one-way requests, reliable two-way requests-reply transactions.
- Optional user-to-user reliability - WTP user triggers the confirmation of each received message.
- Optional out-of-band data on acknowledgements.
- PDU concatenation and delayed acknowledgement to reduce the number of messages sent.
- Asynchronous transactions.
3.3.4 **Wireless Session Protocol (WSP)**

The WSP protocol provides a consistent interface for a connection-oriented and a connectionless session services for the application layer. The connection-oriented service operates above the transaction layer protocol WTP. The connection-less service operates above a secure or non-secure datagram service (WDP). The connectionless service is useful for applications that do not care about confirmation and do not need reliable delivery of data. WSP consist of services for browsing applications (WSP/B). WSP is optimized for low-bandwidth bearer networks with relatively long latency. WSP/B allows a WAP proxy to connect a WSP/B client to a HTTP server.

3.3.5 **Wireless Application Environment (WAE)**

WAE is an application environment that is based on a combination of Mobile Telephony technologies and the World Wide Web (WWW). The purpose of the WAE is to establish an environment to build applications and services. WAE includes a micro-browser environment, which contains:

- Wireless Markup Language (WML) - a markup language that is optimised for use in hand-held mobile terminals. WML has similarities with HTML.
- WMLScript - a scripting language.
- Wireless Telephony Application (WTA, WTAI) - telephony services and programming interfaces.
- Content formats - a set of data formats, including images, phone book records and calendar information.

3.4 **HOW WAP IS INTENDED TO WORK ABOVE BLUETOOTH**

There are four key types of information that are required in order that two devices can use WAP protocols over Bluetooth links. First, the devices must be able to recognize each other as being WAP capable, giving some indication if they are able to act as client or proxy/server. Next, the devices should be able to exchange a short text description of the type of service provided by each; for example, “Copenhagen Airport Information Service”. Next, each device must indicate that it supports the baseline protocol stack as defined by the Bluetooth Specification version 1.0. The current baseline is “LAN Access using PPP”. Finally, the proxy/server should indicate if is has overridden the default UDP port assignments for the various WAP protocols. By providing some basic information, each device can be certain that communication complying with the WAP specifications is possible between the devices.

In the future there will likely be more adaptations of the WAP protocols to the Bluetooth environment. The current specification establishes only a baseline for interoperability purposes. In particular, the choice of the User Datagram Protocol transported over the Point-to-Point Protocol was logical given the widespread use of PPP in WAP implementations over mobile networks. The combination of WAP and Bluetooth technologies provides access to a range of applications that would otherwise be impossible in a wire-based environment. The client-server model for WAP provides highly constrained devices the
ability to access a broad range of information and information services. Allowing scripting capabilities that are independent of the implementation of the device enhances this type of access even further. These abilities in combination allow Bluetooth-enabled devices to far exceed the capabilities of their wire-bound counterparts.

Bluetooth functionality is very much based on the different usage models, which formed the basis of the specification work done by the Bluetooth SIG. These usage models have also had a great impact on the Bluetooth protocol architecture and individual protocols belonging to it.

In designing the Bluetooth protocol architecture, especially its higher layer protocols, the principle was to maximize the reuse of existing protocols for different purposes instead of reinventing new protocols. Protocol re-use also helps software vendors to adapt existing applications to work in the wireless Bluetooth environment, as well as facilitating smooth operation of the applications and their interoperability.

Different applications may run on top of different application protocols. Consequently, most applications use different sets of protocols to achieve the desired functionality of the Bluetooth usage models. Nevertheless, each one of these different sets of protocols (i.e., vertical slices of the whole Bluetooth protocol stack) use a common Bluetooth data link and physical layer. Additional vertical slices are for services supportive of the main application, like SDP (Service Discovery Protocol).
4 PORTING TO WOSP

One of the major parts of work in this project is since Axis’ Bluetooth stack runs under Linux it needs to be ported to eCos. There are two opposing requirements on this porting. First, the memory footprint should be kept small. Second, the changes should be as small as possible to make the porting of a new release easier.

Another part of the project is to find a UDP/IP/PPP stack that could be used within eCos otherwise a porting to eCos is needed.

Finally a WAP stack must be found and ported to eCos and the WOSP requirements.

4.1 PORTING AXIS’ BLUETOOTH STACK

4.1.1 Short Description of the Stack

Axis’ Bluetooth stack is far from a fully featured Bluetooth stack. One should see this stack as a framework; most signals are created, sent and received. The stack currently works in kind of a static way; no true configuration is done yet between peers during a connection. The development of the stack has focused on the Serial Port-profile and the LAN-profile. It is possible to connect two devices and send data or start PPP and then for example run ftp. No other profiles are supported at the moment.

One nice feature with the stack is the modem emulation option. With this option enabled you can connect two computers running Linux with a serial cable and then run the stack without any Bluetooth hardware. This option makes it easier to evaluate the stack.

No layers are completed but the basic functions exist and work, for example L2CAP and RFCOMM does the segmentation and reassembly operations and HCI performs the most necessary commands.

When it comes to SDP there is some really interesting work going on inside Axis, it is expected that in next release there will be some kind of XML server solving the tasks of SDP.

One has to realize that any open source product will never reach completeness since there always will be persons adding new functionality.

Axis’ stack makes use of the kernel in Linux; it works as a driver towards which applications can connect with a “TTY” which is an abbreviation for "Teletype".

When using Linux the computer considers each serial port to be a "device". It is sometimes called a terminal device since once upon a time terminals were the common use for the serial port. For each such serial port there is a special file in the /dev (device) directory. /dev/ttyS0 is the special file for the serial port known as COM1 in the DOS/Windows world. The S in ttySx stands for serial port.

When installing Axis’ stack one also creates a new “device“ with the mknod-command which special file is named ttyBT0. In this way any application that
can communicate with a “device” can make use of Bluetooth. That means that there is nothing to be changed or added in the application when changing from using a serial cable to using Bluetooth.

Figure 4.1 shows how the tty is intended to work. This picture is given in the file Bluetooth.c.

![Diagram of PPP over serial (Standard way)](image)

**PPP over serial (Standard way)**

- User application (pppd)
- **tty**
- **tty driver**
- **UART**

**PPP over Bluetooth**

- User application (pppd)
- "ttyBT0"
- **tty**
- **tty driver**
- **BT stack**
- **ttyS0**
- **tty driver**
- **UART**

![Diagram of Axis’ Bluetooth Stack in Kernel Mode](image)

*Figure 4.1: Axis’ Bluetooth Stack in Kernel Mode*

It is also possible to run the stack in user mode. In user mode no ttyBT is created and new glue functions on the top of the stack is needed. The port to eCos wills very much use the user mode version of the stack. But still the user
mode stack uses the standard ttySx features of Linux. In user mode a receive thread is created that listens (performs a select) to the ttySx. When something is received on the port the thread will be awakened and it will pass the received data up to HCI.

When an application is sending HCI-commands to the Bluetooth module there always comes a response back, a command complete. This response includes information about the command’s success or failure in the Bluetooth module.

Some commands need to be performed before the application can go on, for example the reset command.

These commands make use of Linux’s wait_queues, the sending process blocks after sending. The receiving process releases it when the corresponding event occurs. Before blocking the sender a timer is started that will prevent deadlock if a response is never received.

This concept of blocking is also used in higher layers, especially when setting up a connection and when doing disconnect.

4.1.2 Modifications to Axis’ Stack

Unfortunately there is no support for tty:s in eCos. In eCos there is just support for simple serial drivers, which are interrupt driven and for example include the functions:

- Cyg_io_read - Receive data from the device.
- Cyg_io_write - Send data to the device.
- Cyg_io_set_config - This function is used to update or change runtime configuration of a port. The changeable parameters are word length, baud rate, number of stop bits, number of parity bits and if “hardware handshake” CTS/RTS should be used.

The drawback with the cyg_io_read function is that the length of the data to read must be passed as a parameter to the function. Since there is no way to predict how many bytes that will be sent from the Blue tooth module some kind of workaround is needed.

The serial device driver was modified in this project to be able to wake a process when data from the Bluetooth device arrives at the serial port. This is done with a semaphore. The semaphore’s counter is increased each time a byte arrives. The receiving process stays in an infinite loop doing cyg_semaphore_wait on this semaphore all the time. When the process is not blocked it reads the buffered data, with the semaphore’s counter as the length parameter. The receiver then passes the data to the stack by a call to the function bt_receive_lower_stack, which will pass the data on to the function hci_receive_data. In HCI the wait_queues have been replaced with eCos’s semaphores and the timers have been replaces with eCos’s timer functions.

In bt_receive_lower_stack the overflow bit in the processors line status register (LSR) will be checked. If this bit is set it means that data has arrived faster to the shift register than the processor could retrieve it. If an overrun error has occurred an added variable in HCI named hci_error will be incremented. This variable will also be incremented each time a packet that could not be parsed is received in HCI. The most common reason to an overflow is that a call to printf has blocked to entire processor. These calls to printf are only done in debugging purposes.
With this changes the WOSP stack looks very much as the original stack in user mode.

The changes inside the Bluetooth stack should be kept as small as possible so future updates from Axis could be used with as small efforts as possible. Actually there have been new releases during the WOSP project and all of them have been ported to eCos with little effort. The part where the major changes have been performed is in the bluetooth.c file. This file acts as glue to upper applications but also to Linux as shown in figure 4.2. Both the functions used when transferring data to and from the Bluetooth device as well as to and from the application are placed in this file.

"Glue Layers for stack -

"Application" (ttyET0 in kernel mode)

bt_receive_top() | bt_write_top()

The Stack

bt_receive_lower_stack() | bt_write_lower_driver()

"Physical driver" (serial driver)

Figure 4.2: Overview of the glue functions in bluetooth.c

In bluetooth.c all functions that has to do with tty:s have been removed and the glue functions have been modified to not use tty:s. All changes are surrounded by #ifdef _WOSP #endif, except in the bluetooth.c file, this will make the changes in the stack easy to detect and at the same time they will be kept minimal. To run the new stack –D_WOSP needs to be defined in the makefile.

Since WOSP runs as an embedded system there should be no malloc:s. Therefore an eCos’s memory pool has been added from where all calls to malloc now allocates memory. The size of the memory pool has been set to 4 KB.

One major problem was discovered during the port. This problem comes from GCC’s way to treat layered structures. The problem is that any field inside the structure will be zero padded to use more bits than assigned when the structure is put inside another. In Bluetooth for example in RFCOMM and L2CAP there are several structures, which are defined with fields of just a few bits. An example will show this problem.

If the structure my_struct is defined like
typedef struct my_struct{
    uint32 var_A;
    uint16 var_B;
    uint32 var_C;
    uint16 var_D;
} my_struct;

typedef struct my_layered_struct{
    uint8* data;
    uint32 length;
} my_layered_struct;

And the uint32 is defined on this machine using four bytes. When the data pointer in my_layered_struct is set to point on my_struct. Then the two variables using just 2 bytes will be zero padded to use four bytes as well. So when serializing the structure before sending, the data part will look like this in memory.
| var_A (4 bytes) | var_B (4 bytes) | var_C (4 bytes) | var_D (4 bytes) |

The expected is.
| var_A (4 bytes) | var_B (2 bytes) | var_C (4 bytes) | var_D (2 bytes) |

Since the key idea of every communication protocol is that the packets are well defined the communication will be broken due to this compilation problem. This problem appeared to many on the mailing list at this time and with the help of each other a solution was found. The solution is a command to GCC that will force the compiler to allocate memory as defined. The structure in the example looks like this when the problem is fixed.

typedef struct my_struct{
    uint32 var_A;
    uint16 var_B;
    uint32 var_C;
    uint16 var_D;
} __attribute__((packed)) my_struct;

typedef struct my_layered_struct{
    uint8* data;
    uint32 length;
} __attribute__((packed)) my_layered_struct;

The Axis’ Bluetooth stack has three major drawbacks.
1. The stack is not ready yet. A limited version of SDP is implemented.
2. The stack is not designed to be easy portable to other operating systems and other machines. For example functions switching between big and little endian are not used. And the structure problem makes the stack sensitive to which compiler used to build it.
3. There is no documentation except the source code.
4. It is not certificated.

The good news is that Axis effort right now is to extend SDP to use a XML server and that several patches have been sent to the list to solve the structure and endian problems. Right now there is also work going on in the HCI layer to support multipoint communications.
In WOSP the downloaded size of the Bluetooth stack compiled using –O2 was approximately 71 KB, RAM mode with GDB support. The details from the porting can be found in Appendix C.

4.2 PORTING UDP/IP/PPP STACK

As mentioned the protocols UDP, IP and PPP should be added on top of Bluetooth. Since eCos ships with a TCP/IP stack and this is ported to the operating system this formed the basis of the solution. One drawback of including this stack in the v1.3.1 is that the net package has been in a heavily evolution phase due to discovered bugs reported to the mailing list. This means that the package included in this release is out of date. The stack was heavily modified to meet the memory requirements of the system. First of all TCP, Ethernet drivers, FTP etc was removed since only UDP is used as a bearer of WAP.

The stack was also modified to run as program on top of eCos and not as a driver in eCos. The PPP-layer is not yet part of eCos net package since it is intended to make use of Ethernet. PPP will eventually be added in later releases. The if_ppp.c file was downloaded from http://www.openbsd.org. In this file some changes had to be done to be able to compile it under eCos. Also a block of #ifdef _PPP #endif was inserted in the output-routine to be able to pass the data to Bluetooth.

Since the stack works above Bluetooth there are no need at the moment to offer routing capabilities. The application in this project is intended to exchange information from one Bluetooth device to another. With Bluetooth a connection is first set up and then data is transmitted. Thus when a connection is established the IP-packets know where to go. In ip_output.c a new block #ifdef _PPP #endif was inserted to avoid all the routing stuff.

The whole networking stack of eCos originally descended from OpenBSD. Most stacks descend from OpenBSD and FreeBSD. FreeBSD is the operating system most of the key devices out on the Internet runs. Theses devices make the Internet possible. Perhaps most famous example right now is that Microsoft runs FreeBSD on their Hotmail servers.

But when putting the modified stack on top of the Bluetooth stack the memory showed to be too small. The reason was that both eCos, Bluetooth and the stack was compiled in debug mode and on top of that is all the debugging in this project done in RAM space. The advantage of running the whole program in RAM is that no device is needed to connect via the JTAG on the data- and address-buses, this kind of device costs several thousands of dollars.

4.3 MEMORY EXPANSION

Now the only way to solve the problem was to extend the memory on the AEB-1. The original 256 KB of SRAM was expanded by another 512 KB. Two new 512x8 SRAM were piggybacked on top of the old ones. Every pin except four could be leaded on the corresponding pins on the old memories.
Then the work of setting up the ARM7 processor to use this new memory started. This is a minor problem if the memory could be set up during runtime as in the test program shown in Appendix D.

But when running eCos a new GDB stub needs to be flashed into ROM. This stub is used during the download process to place the different parts of the code, e.g. data and text, in different parts of memory. To set up memory an assembler script needs to be written. Then before the new memory can be used eCos needs to be rebuilt with the new memory sizes otherwise there will be an error already during compilation.

The procedure to create a new GDB-stub is well documented in eCos documentation so there should really be no trouble. But when creating the stub to AEB-1 revC a bug in a link script in eCos sources where found. This bug makes the communication with the board impossible. Instead one should use the files used when creating a stub to revB. The whole procedure used when expanding the memory on AEB-1 and configuring eCos to use the new memory can be seen in Appendix D.

The new memory worked fine for several versions of the WOSP sources. But a strange bug appeared when a new process was added. Every packet received from the Bluetooth module contained at least one error, it was one byte missing not erroneously received but just missing. The byte missed in the packet was always number three or number four, just after the HCI header. It turned out that the new process made the receivers buffer to be placed in the new memory. The new memory was configured to use seven wait states while the old one used only one. When a packet is received on the RS-232 every byte is read into a shift register and then an interrupt occurs that will empty the register and place the byte in a buffer. The interrupt in eCos are split into two levels, an ISR not possible to interrupt and a DSR that is possible to interrupt with an ISR. The documentation from eCos regarding ISR’s and DSR’s is quoted in Appendix E.

In the serial package it is the DSR that moves the received byte from the shift register to the buffer in memory. When the buffer was placed in the new slower memory and the RS-232 used 57.6 kbit/s an ISR occurred before the memory writing in the last DSR was ready.

This error was removed when a new stub configured with fewer wait states on the new memory was flashed into ROM. The number of wait states in the new stub was calculated like # waitstates /processor speed > memory access time, which gives 2/24*10^6 = 83.33 nsec > 55 nsec.

### 4.4 PORTING THE WAP STACK

An open source WAP stack was needed in WOSP. The first option found was the Kannel project at [www.kannel.org](http://www.kannel.org), which is a very active project with several users. The drawback found with this project is their assumption of running the server on very powerful computers. Another drawback was found when a question about the security layer, WTLS, was seen on the mailing list. A problem with patents on the algorithms used in WTLS was mentioned. The owner of these patents is RSA. These algorithms, RC5 block cipher and the RSA key exchange algorithms, will never be part of Kannel, due to legal issues, since it is an open source gateway.
4.4.1 Short description of Ophelia

Another open source gateway named Ophelia done by www.3ui.com was found. From their experiences when building the WTLS layer for the current Kannel architecture, they could identify some of the areas in the current system that needed improvement. They found out that the architecture of Kannel was suffering from some problems. Advantages of Ophelia over Kannel are:

- With Kannel each new transaction from clients spawns a new thread, which means very memory consuming and thus not a good design in an embedded system were one want to have control of the threads.
- Ophelia is event (message) driven at the individual WAP stack layer - each layer is not polling but "blocking"/"sleeping" thread until awakened by the message events.
- WDP layer Supports only UDP bearer (datacalls), no SMS bearer support yet. Which is just what this project needs and thus limiting the work to port it.
- Kannel supports WAP v1.0 and Ophelia supports WAP v1.1. Ophelia is also an open source gateway, which means that due to legal issues, codes for supporting some algorithms (RSA and RC5) cannot be made open source. But there is a commercial version available which supports these algorithms.
- Ophelia ships with extraordinary documentation, much of the following parts can be found there.

The biggest drawback seems to be that 3ui.com has left the open source community. The latest two version of the gateway are not available as free open source.

4.4.1.1 Architecture of Ophelia

The fundamental change proposed by Ophelia architecture is the incorporation of the entire WAP stack within one single daemon in the user space. This daemon is designed to then spawn layer threads as and when the respective layers are invoked. In addition to these, the implementation also includes a timer thread.

Once the WAP stack daemon is started, the WAP_box process creates one instance of all the necessary WAP stack layers as threads as well as a time thread. To draw a rough analogy, the WAP_box thread acts like the bearer to the rest of the WAP stack threads or layers. What this means is that, the WAP_box is the thread, which listens for UDP packets on all the "well-known" WAP ports viz., 9200, 9201, 9202, 9203 etc.
- Port 9200 corresponds to WAP connectionless session service WDP/WSP.
- Port 9201 corresponds to WAP connection oriented session service WDP/WTP/WSP.
- Port 9202 corresponds to WAP secure connectionless session service WDP/WTLS/WSP.
- Port 9203 corresponds to WAP secure connection oriented session service WDP/WTLS/WTP/WSP.

The timer thread on the other hand provides the "time-out" and "wake-up" (meaning a message is in the queue) alerts for the WTP thread.
4.4.1.2 Operational Procedures

Once a client device initiates a transaction, message packets start arriving at any of the above-mentioned ports, depending on the type of connection requested by the client device. These packets are then forwarded to the WDP thread as "T_DUnitdata" primitives or messages which will then further be processed and routed to the appropriate subsequent layers in the form of a "T_DUnitdata" primitives or messages.

For example, a packet coming in on port 9200 will be processed, packed and forwarded to the WSP thread. The reply coming back and meant for the same tuple will be eventually forwarded back to the WDP thread through the same thread.

The forwarded packets will generally have the following structure -
- Source address (unsigned integer: 32 bits)
- Source port (unsigned integer: 16 bits)
- Destination address (unsigned integer: 32 bits)
- Destination port (unsigned integer: 16 bits)
- User data (len: unsigned integer 16 bits, data: pointer to unsigned 8 bit char)

4.4.1.3 Layer Communication

Communication between the main program thread and the different WAP layer threads, primarily, takes place via primitives mentioned earlier. Each of the WAP layer threads i.e., WDP, WTLS, WTP and WSP, has its own event queue for any such primitives meant for any of the WAP stack layer threads. Likewise, the timer thread also has its own timer event queue.

Generally whenever a stack thread is created, without any message event, they are blocked. However, when a message event is forwarded by the bearer (WAP_box) to any of the WAP stack threads, the event is queued into the particular layer thread's queue. This will be done by the function "Eventadd()". The recipient thread will then be awakened and it will pull the event message out of it's queue using "Eventget()". Following this, the fetched message will then be processed and the destination and source addresses and the ports are taken into account to examine if the thread has an existing "state" to handle this event. If it detects the absence of one, a new "state" will be created to handle the message.

Ophelia is a WAP gateway; even if it won’t be used as such in this project there is a need to explain how it is intended to work. Right at the other end of the flow of the event messages mentioned above are the HTTP threads. The HTTP threads are spawned by the WSP thread. One such HTTP thread is spawned for each HTTP request. These HTTP threads then take over the responsibility of actually retrieving the content from the web servers or proxies as the case may be. Subsequently, the content is compressed and converted with the help of an XML compiler. The converted
content is then passed back to the requesting WSP thread to be queued as an event message in its queue.

Each of these HTTP threads would be deleted once the thread has passed the processed and converted data back to the requesting WSP layer thread.

Figure 4.3: Design of Ophelia

Figure 4.3 shows how Ophelia is designed to work. The picture contains:
- WAP connectionless session service (protocol WDP/WSP at 9200 port - those dotted lines) and
- WAP secure connectionless session service (protocol WDP/WTLS/WSP at 9202 port - those solid lines).
- Flow of data streams (message events) are marked with arrows.
- For WAP session service, the sequence of flow is numbered.

4.4.2 Modifications to Ophelia’s Stack
In WOSP a WAP-server should run on top of Bluetooth. This means that the gateway must be modified to not make any HTTP requests but instead return data from the embedded system. In this first version of WOSP a limitation has been done; data will be transmitted using only WAP connectionless session service using WDP and WSP. If additional features such as security are needed in the future the excellent architecture of Ophelia will make this a moderate task.
When a packet arrives to the WAP_box it is passed up to WDP, which passes it on to WSP. In WSP everything is just as in the gateway case except when it comes to the creation of a HTTP thread. Instead of creating this thread an internal WML page is fetched and returned back to the WSP. The modifications to Ophelia are very small. The only functional change is that any HTTP request is never done.

Some other modifications have to be done to port the stack to eCos. Most changes come from Ophelia’s use of the standard Linux thread library, pthread. A wrapper was created to avoid changing every call to pthread-functions from the WAP-stack. This wrapper-layer redefines the pthread-functions to thread-functions defined by eCos.

A variable size memory pool has been added. Every call to malloc in the WAP-stack uses this pool. The functions on the memory pool and the size have been specified in wrapper.h. This size is currently set to 16 KB but at least 10 KB out of these 16 KB are always free in this version of WOSP.

The downloaded size of the WAP layers compiled using –O2 added approximately another 61 KB. That makes the total downloaded amount in WOSP to stay around approximately 160 KB.

The details about the porting can be found in Appendix F.
5 REDESIGN OF WOSP

One of Combitech Systems’ goals of this project was to evaluate the Bluetooth and WAP open source components. Another goal was to identify products where these techniques could be used. There were never any doubts about the usefulness of the project. Apparently this technique would be needed as embedded systems are expected to open up to the rest of the world. In the requirements of the project one device running a WAP-client over Bluetooth should be able to show WML pages stored in another device running a WAP-server over Bluetooth. This was actually needed in some projects and became the main use case, just a client directly connected via Bluetooth to the server.

The current Bluetooth profile supporting WAP contains, as was shown above, not just WAP and Bluetooth but also UDP, IP and PPP. Those layers make it possible to address any server connected to Internet containing WML pages. One use case could be that a phone or other another device running WAP browser connects to the Internet via Bluetooth. That is a mobile phone or Palm connects to a LAN access point using Bluetooth. But why should you be surfing on your phone when you are in the office?

Another and more useful idea is that in the other end, on the server side, Bluetooth is used. In that case the server connects to Internet via Bluetooth. The server then sends its packets via Bluetooth out on the Internet to access any device connected, probably a WAP-gateway.

To see the usefulness of such a server imagine that your car contains a WAP-server, which is given an IP-address, and when you park the car in a parking house the owner offers your car a connection to Internet. The access between your car and the access point goes via Bluetooth. Then it is possible for you to get access to information stored in the WAP server in your car from any place in the world using the UDP/IP protocol. An additional feature could be a webcam placed in your car. Well, maybe not the best example but one can think of several use cases.

Another use case is when the WAP server is connected to a GSM modem via Bluetooth and UDP/IP is running at top of GSM. Then the server needs to support the LAN profile as stated in [Bluetooth].

5.1 REASONS TO THE REDESIGN

Anyway Bluetooth’s profile does not fit into the use cases identified during the WOSP-project by several reasons.

1. The intension of WOSP right now is not to be connected to Internet. One reason is the memory requirements and another is the limitation of the project.

2. When running WAP between a client and a server that are connected before the first packet is sent there is no need to be able to address millions of devices. The direct Bluetooth connection between the two devices determines where the packet will go.

3. The main use case during the project didn’t need any routing capabilities.
One can think of several situations where a device containing a small WAP server such as WOSP running on top of Bluetooth would be suitable. Examples of use cases are wireless wallets and when a serviceman should fix a machine that is out of order or just performs some services. The serviceman wirelessly connects to the machines’ built in WAP-server and retrieves some key parameters. Hopefully these failure codes will give him a hint on what is wrong or at least in which module of the system where the failure is located. This makes it possible to replace cables and service tools with a standard mobile phone or PDA.

5.2 CONCLUSIONS OF THE REDESIGN

This means that the profile will not be followed and that WOSP will not fulfil the requirements on a Bluetooth device. It still means that the embedded system will be opened up to the rest of the world. The time spent during the project when porting UDP, IP and PPP to eCos is thrown away but this limitation on the project will certainly make the goals reachable within the deadline.

Hopefully there will be mobile phones in the near feature that runs WAP directly on top of Bluetooth. As was stated in section 3.4 there are probably emerging new profiles supporting WAP over Bluetooth. A mail containing a question of today’s WAP over Bluetooth solution and a suggestion of a new profile with WAP directly above Bluetooth was posted on the WAP Forum mailing list. No serious answers against such a profile were seen but mostly answers that defended the current solution. There is no doubt about the usefulness of the current profile but maybe there is or will be a need for a lightweight solution.

All mobile phone and PDA users having access to a non-cost browsing tool, that WAP over Bluetooth is, would certainly strengthen both WAP and Bluetooth and perhaps even stake out the path to a new market.

The market of devices with embedded systems that would benefit from being opened up to the world with the possibility of running WAP directly over Bluetooth couldn’t be negligible.

One more aspect is that when just opening up to devices located in the short range of the server not using IP brings forth that the security problems of today’s Internet could be bypassed. The built in security of Bluetooth and WTLS would do the job without any kind of firewall being necessary.

The biggest positive result of the redesign to this project is the lowered memory consumption, about 100 KB lower.

The major drawback is that these removed protocols had more features than just routing packets. WDP makes WAP bearer independent but it demand complete data packets. Since the default size of RFCOMM packets is 127 bytes any WAP packets exceeding this size will be segmented when they arrive to the other side. Thus there is a need for reassembly functions above Bluetooth. This function has been added to WOSP.
5.3 WOSP’S DEMO SYSTEM

The major effort in this project has been laid on the server side, which is the part of the system that resides and contains information in the embedded system. To be able to show the system a demo system has been set up. In this demo the client should to be able to show the information in a nice graphical and natural way, as a mobile phone will display it. The feeling should be like using a phone.

To meet these requirements a phone emulator of Ericsson’s R380 phone was downloaded. The emulator contains a WAP browser, which is used to browse the WML-pages in the WAP-server. This phone runs Epoc as operating system. The drawback of this really nice emulator when used in this demo is the fact that it needs Windows. No emulator that uses Linux could be found from any mobile phone manufacturer.

The drawback comes from Axis’ Bluetooth stack using Linux. Since another porting of the stack to Windows was not within the plan of the project another solution had to be found.

The best solution would have been an open source WML browser written in C or C++ running on Linux. Then real client software designed for WOSP could have been created. Only one intention to make such a WML browser was found, http://www.wmlbrowser.org/, but this project was not apprehended as being alive.

Now the client consists of two computers one PC running Linux and another running Windows. This configuration is just used in demo purpose and should not be taken as any sort of normal use case. They are just used to make the client application really easy to develop and equipped with a nice graphical interface.

The phone emulator runs on the Windows machine. This emulator connects directly to either a LAN or a dial up connection and sends its packets to port 9200 (WDP/WSP). In this demo the emulator will send its packets via the LAN to a computer running Linux. On the machine running Linux the WAP_box process from Ophelia will receive the packets on port 9200 and then pass them on to bt_receive_top in Axis’ Bluetooth stack which is designed to run on Linux. The stack is run in user mode, which makes it able to debug.

No other layers from Ophelia than the WDP is used on the Linux machine, which can be called a gateway since it retransmits packets received on the LAN to Bluetooth and the other way around. No modifications have been done to the Bluetooth stack on this machine. But some features have been added. First a HCI command setting the page timeout and another HCI command setting the connection timeout have been added to the menu. Second a command called wap_box has also been added. If the value one is given with this command the WAP_box process and the WDP_machine will start. If the value zero is given they will be deleted.

With the max data size being 127 bytes from Bluetooth a reassembly function is needed in the WAP_box before the data is passed back to the phone emulator.
Figure 5.1: System overview.
5.3.1 System overview

The first thing that has to be done is to set up the RFCOMM-connection between the Bluetooth devices and then to start the process that will receive requests on the LAN and retransmit them on the Bluetooth link to WOSP. This is done from the Linux machine with the following commands in bluetooth/apps/userstack.

```bash
./btd  -i ericsson -r client  Starts the program in user mode.
inq     Starts an inquiry. Wait for the response.
Wap_box 1    Starts process listening to port 9200.
```

Then the emulator running on any computer connected to the Internet using Windows need to be configured with the Linux computer’s IP-address as the address of the WAP-gateway. The demo starts when an address is entered on the phone, e.g., www.wosp.nu/index.wml.

1. The request will arrive on port 9200 on the Linux computer since Ericsson’s R380 runs in connectionless insecure mode. The WAP_box process will receive the UDP packet and create a WDP-packet out of it (a T_DUnitdata object). WAP_box sets up the ports that are dedicated to WAP traffic, 9200-9203, and then it performs a select on these ports. Select is an interrupt driven function that makes the process sleep until a packet is received. WAP_box as well as the structures used in a WDP-packet and the UDP receive- and send-functions are the same one as used in the Ophelia WAP-gateway, which was described above.

2. The WDP-packet is serialized and then sent to the Bluetooth stack as data on one specific RFCOMM channel. RFCOMM, L2CAP and HCI will add their headers to the data. The maximum size of the data is defined in the file bluetooth.c. This restriction in size comes from the WOSP server in which the buffers used to save data now is statically allocated. In RFCOMM the data is segmented to packets of 127 bytes.

In L2CAP the data can be segmented into several packets small enough to be passed by HCI to the Bluetooth module. The default packet size in L2CAP is 672 bytes. The RFCOMM packets must not be greater than this.

3. The Bluetooth module receives the packet on the serial cable. It removes the HCI-header.

4. The data is passed over the air to the other Bluetooth module.

5. This Bluetooth module adds a new HCI-header and sends the packet on the serial cable to AEB-1.

6. The packet generates an interrupt in eCos and the semaphore is incremented as each byte arrives. The receiver process awakens from the semaphore and passes the received bytes on to the HCI-layer. This layer waits to pass the packet on to L2CAP until all bytes in this HCI-packet are received. L2cap waits on all L2CAP packets until the RFCOMM packet is passed further on to the RFCOMM channel. RFCOMM puts the WDP-packet in the WDP_machine’s in-queue possibly after some reassembly if the packet was larger than 127 bytes.

7. The WDP_machine receives a WDP_event and processes the packet. The packet is passed on to the correct layer in the WAP-stack. Since this demo uses insecure connectionless mode the packets is sent to WSP-layer.
8. The WSP_machine receives a WSP-event and finds out that it is a connectionless packet and then parses the request.

9. The request is sent to the WAP-server function, which finds out that the WML page for example index.wml is requested. Necessary variables are fetched and included in the page before it is compiled.

10. The compiled WML page is returned to the WSP_machine.

11. Source and destination addresses are swapped and a new packet is created. The new WSP-packet is put in a WDP-event and passed down to the WDP_machine.

12. When the WDP_machine receives the event it finds out that the packet should be sent downwards. It serializes the packet and sends it as data to the specific RFCOMM channel that is used.

13. After the response has passed the Bluetooth stack it is passed on to the Bluetooth module on the RS-232.

14. The module sends the possibly fragmented message through the air to the master module.

15. When the Bluetooth-packets have reached the ttySx it is fetched by the receiver process on the Linux machine and sent to the stack.

16. All the Bluetooth headers are removed from the packets and they are reassembled before the massage is passed on to the UDP layer, which sends it further on through the stack, IP and Ethernet headers are added to the WAP-packet. The packet is then sent via the LAN to the computer on which the phone emulator is running. The emulator parses the packet, sends it through the WAP layers, decompiles it and finally shows the WML-page on the screen.

---

**Figure 5.2: Main page on WOSP**

**Figure 5.3: Index page on WOSP**
With this page the four LED’s that are located on the AEB-1 can be lightened or turned off. The options are highlighted in this picture.

With this page one can select from which of the two memory pools that the size of the available memory should be returned. All calls to malloc allocate their memory in these pools.
The processors load can be displayed with this page. Andrew Lunn at Ascom, Switzerland, sent the source code of this load monitor to eCos’ mailing list. The load in a real-time system should never be 100 percent but in this system it is while GDB is doing debug prints on the screen.

This page shows the error variable in HCI, which was mentioned in the porting section. There is also an option that makes it possible to set this variable to zero.
6 OUTLOOK

This project has shown that open source can be used to develop embedded systems. WOSP used an open source toolchain from GNU and an open source operating system from Red Hat. How an embedded system can be opened up was also shown in this project.

Perhaps the most useful output from the project was the fact that it is possible to run a WAP server over Bluetooth within a very small memory. The whole software could be downloaded within approximately 160 KB (RAM mode).

The software developed needs to mature before it can be used in something more than a demo.

Bluetooth is a hot spot in today’s hysterical mobile internet era. Combitech Systems and WOSP can benefit from this fact. There are extremely many developers out there that want to start elaborate the Bluetooth technology and cause of that having the will to contribute to WOSP. Perhaps the best way to go on with WOSP is to make it open source. If sharing it with both the eCos and the Axis’ Bluetooth community the idea would certainly evolve rapidly. Several developers have already asked if this project will be made open source, even Red Hat has asked.

In eCos perspective it would be a great feature having a Bluetooth stack. The next step taken to integrate the stack with eCos will be to create a component, which can be loaded into eCos using the package tool shown in figure 2.2.

But before the Bluetooth stack can be used in any commercial product it needs to be further developed. Keeping the stack up to date with the new releases from Axis is necessary.

The advantage of open source to Combitech Systems is that much of the work to get a stable and fully featured Bluetooth stack ported to a license-free open-source operating system will be done by the community. Another advantage is that the company will be known around the world among the open source people.

Since there is a great activity of porting eCos to new processor architectures it is no wild guess to expect that the stack will be made more architecture independent. This brings forth that all developers cooperating will solve that kind of problems little and big endian architectures are causing. And the tests of interoperability will be rigorous.
## 7 REFERENCES

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
</table>
8  APPENDIX A
8.1  QUESTION AND ANSWERS ABOUT eCos

Question posted at eCos mailing list:
On Tue, Nov 28, 2000 at 10:49:51PM +0800, anarchy wrote:

Our company is trying to find a "serious kernel" to get rid of pSOS and VxWorks. I found eCos very charming but have no experience using it. Is here any company using eCos longer than one year and thinking it's good(or very good or bad). I need to hear some serious comment about using eCos by senior users. Please give me any advice or suggestions. Thanks a lot.

Answer 1
We've been using it for about a year now (we've been shipping a product that uses eCos for about 4 months). I've got no complaints. The level of support provided by the mailing list is _far_ better than support my colleagues have gotten for pSOS.

My experience is that development under Linux/Unix involves fewer hassels than under Win32 (Cygwin/toolchain issues are solvable but ever-present).

--
Grant Edwards
grante@visi.com

Answer 2
On Tue, Nov 28, 2000 at 12:44:02PM -0500, Lewin A.R.W. Edwards wrote:

> I'm curious: What specific platform do you use for development, Linux > (Intel). and what is your target CPU/architecture?
ARM7TDMI (custom board using a Samsung KS32C5000).

> The reason I ask this is because I have one very major gripe with the free > open-source level of support from Red Hat, and that is that there is no > version unification, no known point from which to start.
I'm not sure what you mean. There are "official" releases: the current one is 1.3.1, and you also have access to CVS sources. Commercial products only provide the official releases.

> and every time one hits a problem one has to start debugging it from the > ground up.
Are you saying that you would prefer that official releases came out more often?

> I'm wondering if I have experienced my usual Murphyesque ill luck and > chosen the one maverick platform (pun intended) in the "supported > platforms" list,
Don't know. I've never used a supported platform.
but my experience with eCos and its toolchain thus far is that almost no component has installed/configured/compiled per the installation instructions, and a big part of the problem is that instead of providing specific snapshots of known-working versions, the install instructions refer to nonexistent historical versions.

I guess I didn't have any memorable problems installing and building eCos. It did take a few days of messing around to get the toolchains built, but certainly no more hassle than I've run into with some of the commercial toolchains I've debugged over the years. The nice thing (for me) about using the gnu toolchain is the lack of a learning curve. I've been using gcc et al for almost 15 years, so it was nice that I didn't have to learn yet another set of compiler, assembler, and linker options when I started doing eCos development.

Point taken about Linux host being less problematic, but I tried both Linux and Cygwin and had only slightly different results. Also, I don't know about other embedded engineers, but it is a significant annoyance to me to have to use anything other than Win9x for development.

I've never used Windows for embedded development, though I used DOS briefly many years ago. I've always found SW development under Unix to be easier than what I've observed other Windows users doing. I started out 15 years ago doing embedded development under Unix, and though I've tried a few other systems in the meanwhile (VMS and DOS), I still find Unix to be the most productive environment.

because most of the special-purpose hardware we use is DOS-only. It's quite painful to have to use two PCs instead of one.

I don't buy DOS-only hardware. Except for that one HC11 emulator. That one I ran under Linux's DOSemu. Worked better than it did under real DOS.

One certainly couldn't describe eCOS as a fast track to anything; there are a dozen different steep learning curves to be climbed before you can even build the OS, much less try to link your own program.

It didn't seem any more difficult than the commerical RTOSes I've used. The process of porting eCos to a new platform and writing drivers for custom hardware seemed equivalent to or better than what I've seen trying to get other RTOSes running on custom hardware.

Perhaps there are other systems that are quicker than eCos if you want an off-the-shelf turnkey system, but I've always done development for custom boards, so I can't make a comparison.

--
Grant Edwards
grante@visi.com
Answer 3
I certainly don’t want to have an argument about the quality of the free distributions or anoncvs, nor of the politics of free software or open source projects, and I’m definitely not disputing what you say. I’m also not the “political officer” “commercial contact” here. Please take this as a personal aside, not an official Red Hat frontman’s comment!

In a nutshell, providing specific snapshots of known-working versions which we have thoroughly tested and that we continue to test for your specific platform, is the business side that allows all this stuff’s existence in the first place. Put another way: we’re more responsive to folks with a support contract - sorry, but that’s the way it is.

...which doesn’t excuse poor quality in free distributions. But this list helps a lot with that, we hope. And for most people who have climbed that learning curve, it seems that newer (and maybe unreliable) is better than more solid but out of date in terms of feature set. Hence the anoncvs service.

Huge

Answer 4
> I’m in a similar boat, we will probably go with one of 3: VxWorks, Nucleus or eCos. I’ve built the kernel (I think!) but now I don’t know what I can do:
> How do I get my own Apps. in?

Look at the Tutorial section of the documentation.

> Can I simulate it on my dev platform (Linux)?

Yes. Look for references to the Synthetic target. It runs eCos as a process on Linux.

> Basically, what do I do?

RTM and try it.

> Undocumented does seem to be the problem: I’ve looked at eCos dev. On Linux, a colleague looked at eCos dev. on Windows. Neither of us have actually got anywhere yet (probably a couple of weeks of combined effort).
> There comes a point when we give up.

I personally have not found the documentation too bad. Its terse, but just about all there. Comming from a Unix background, man pages etc, im used to it and like it.

When i get a new release i can generally compile everything from scratch in half a day. The tool chain takes the longest time. The limit is disk io, not human. I just follow what i says in the documentation, plus one ecosconfig import file to make things a little easier for me.
If you are having realy problems mail to the list. One thing where eCos
difference from VxWorks and Nucleaus etc, is there is a net community, a
bazaar is in operation and people are working on the ego.

Andrew

Answer 5
On Wed, Nov 29, 2000 at 05:43:28PM +0800, Xavier Wang wrote:
> Does eCos have any relationship with Linux?
Yes.
> Can we say that eCos is an embedded Linux?
No.
> How does eCos differ from other embedded Linux?
N/A
> Or furthermore, how does eCos differ from other embedded
  RTOSes?
It's bigger than some, smaller than others, cheaper than most, and open-source.
--
Grant Edwards
grante@visi.com
9 APPENDIX B

First the eCos operating system had to be installed. Information about eCos can be found at http://www.redhat.com/products/ecos/ and http://sources.redhat.com/ecos/. This operating system is highly configurable so the fewer the requirements, the smaller the kernel. A typical kernel that includes a scheduler, memory management, RTC support, several threads, and interrupt handlers would require about 3K ROM and 1K RAM.

9.1 INSTALLING GNU TOOLKIT

Problem 1:
The hints on http://sourceware.cygnus.com/ecos/install-windows.html were followed. The first problem was when making gcc.

```
-DTOOL_INCLUDE_DIR="/"tools/H-1686-pc-cygwin/lib/gcc-lib/arm-elf/2.95.2/.././echo /tools/H-1686-pc-cygwin/| sed -e 's|^/tools||' -e 's|^/||' -e 's|^\[^/||/|' -e 's|/[^/]*|../|g'|arm-elf/include"
-c `echo /src/gcc/gcc-2.95.2/gcc/cccp.c | sed 's,^ \ ./,,'` 
```

In file included from /src/gcc/gcc-2.95.2/gcc/config/arm/aout.h:301, 
  from /src/gcc/gcc-2.95.2/gcc/config/arm/elf.h:354, 
  from /src/gcc/gcc-2.95.2/gcc/config/arm/unknown-elf.h:171, 
    from tm.h:6, 
    from /src/gcc/gcc-2.95.2/gcc/config/i386/xm-i386.h:43, 
      from config.h:8, 
      from /src/gcc/gcc-2.95.2/gcc/config/i386/xm-i386.h:43, 
    from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.2/../../../../include/windef.h:141, 
      from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.2/../../../../include/windows.h:96, 
        from /src/gcc/gcc-2.95.2/gcc/config/arm/unknown-elf.h:226: parse error before `{"' 
    from /src/gcc/gcc-2.95.2/gcc/config/arm/unknown-elf.h:226: warning: unknown escape sequence: `\' followed by char code 0xd
```

The people at the crossgcc@sourceware.com mailing list thought this came from the return characters being wrong (0xd).
To solve this gcc-core-2.95.2.tar.gz was downloaded again. This time the error became:

```
-DPREFIX="/"tools"
-c `echo /src/gcc/gcc-2.95.2/gcc/prefix.c | sed 's,^ \ ./,,'`
```

In file included from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.2/../../../../include/windef.h:141, 
  from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.2/../../../../include/windows.h:96, 
    from /src/gcc/gcc-2.95.2/gcc/prefix.c:69:
  /usr/lib/gcc-lib/i686-pc-cygwin/2.95.2/../../../../include/winnt.h:2062: two or more data types in declaration of `type name'
make[1]: *** [prefix.o] Error 1
```

Solution:
When following the instructions two files will be corrupted, arm.h and reload1.c. Those files have to be corrected with a script. Example for arm.h:

```
$ tr -d '\015' < arm.h > tmp
$ mv tmp arm.h
```
Problem 2:
Some people have had the same problems with returns characters when making gdb, I didn’t. If the same error occur the script in problem 1 will do fine. When installing gdb I had the following error.

```
gec -g -O2 -Wl,--subsystem,console -o gdb.exe main.o libgdb.a rdi=share/libangsd.a ../sim/arm/libsim.a ../bfd/libbfd.a ../readline/libreadline.a ./../opcodes/libopcodes.a ../intl/libintl.a libiberty/libiberty.a `if test -r ../libtermcap/libtermcap.a; then echo ../libtermcap/libtermcap.a; else echo -ltermcap; fi` ./libgui/src/libgui.a -L/tmp/build/gdb/itcl/itcl/win -litcl30 -L/tmp/build/gdb/itcl/itk/win -litk30 -L/tmp/build/gdb/tix/win/tcl8.0 -ltix4180 -L/tmp/build/gdb/tk/win -ltk80 -L/tmp/build/gdb/tc 1/win -ltcl80 -lm -lm ../libiberty/libiberty.a -luser32 -lshell32 -lcomdlg32 -ladvapi32 libgdb.a(gdbtk.o): In function `gdbtk_init': /src/gdb/insight-5.0/gdb/gdbtk/generic/gdbtk.c:433: undefined reference to `Tix_Init' /src/gdb/insight-5.0/gdb/gdbtk/generic/gdbtk.c:435: undefined reference to `Tix_Init' collect2: ld returned 1 exit status
```

Solution:
It’s probably a problem in the way Cygwin produces DLLs. Workaround is to not link with the DLL, but instead link with the static library that is also produced.

```
$ cd /tmp/build/gdb
$ mv tix/win/tcl8.0/tix4180.dll tix/win/tcl8.0/tix4180.dll.old
$ cd gdb
$ make all
$ cd..
$ make -w all install
```

9.2 GETTING eCos RUNNING

When everything is installed as described on the page http://sourceware.cygnus.com/ecos/install-windows.html it is recommended to open the document overview at something like D:\Program Files\Red Hat\eCos\doc\index.html

Getting started with eCos is a good start. First of all an eCos-stub for gdb has to be downloaded to your target, see “ARM AEB-1 Hardware Setup 43”.

Invoke gdb by the command `arm-elf-gdb -nw`

Connect to the target by issuing the following commands in GDB console mode:

```
(gdb) set remotebaud 38400
```
NOTE: The command (gdb) set mips-force-32bit-saved-gpregs should not be used.

Then eCos has to be built matching your processor (AEB-1). This is done in the Configuration Tool (Program->Red Hat eCos-> Configuration Tool). The procedure is given in “Configuring and Building eCos from Source 64”.

Your first program is compiled using this command e.g.
bash.exe-2.04$ arm-elf-gcc -g -I/d/ecos-work/myeCos_install/include hello.c -L/d/ecos-work/myeCos_install/lib -Ttarget.ld -nostdlib
The executable program will be called a.out. If you prefer some other name add the option -o my_\_own\_name.
The commands to run the program look like this.

**Alternative 1 – prompt**

bash.exe-2.04$ arm-elf-gdb -nw a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=arm-elf"...
(gdb) set remotebaud 38400
(gdb) target remote COM1
Remote debugging using COM1
0x401a604 in ?? ()
(gdb) load
Loading section .rom_vectors, size 0x60 lma 0xc000
Loading section .text, size 0xe7d0 lma 0xc060
Loading section .rodata, size 0x3e8 lma 0x1a830
Loading section .data, size 0x2fc lma 0x1ac18
Start address 0xc060 , load size 61204
Transfer rate: 28801 bits/sec, 302 bytes/write.
(gdb) continue
Continuing.
Hello, eCos world!

NOTE: Make sure the paths don’t contain any spaces.

**Alternative 2 – graphical**

bash.exe-2.04$ arm-elf-gdb a.out &

This command invokes the graphical Insight-debugger. To run your program through this interface just select ‘Run’. First you will be prompted to select which target you want to connect to, select ‘Remote/Serial’ and press ok, then you can see when your code being downloaded to the board.
9.3 DOWNLOADING THE PROGRAM WITHOUT GDB

**Note!** When running the program without GDB there should not be any function-calls in cyg_user_start

### 9.3.1 Configuring eCos

First of all eCos has to be configured not using gdb this is done with the following settings in the configuration program.

**Fine grained alterations:**

edit the ecos.ecc file with the following changes

- **Purpose:** Set the TTY console to the device /dev/ser1 (which we'll setup below)
  - **At:** CYGPKG_IO_SERIAL_TTY_CONSOLE
  - **Change:**
    - from: `# use_r_value " \\/dev/ttydiag"`
    - to: `user_value " \\/dev/ser1"`

- **Purpose:** Turn off the TTYDIAG - which I think is to support gdb
  - **At:** CYGPKG_IO_SERIAL_TTY_TTYDIAG
  - **Change:**
    - from: `# user_value 1`
    - to: `user_value 0`

- **Purpose:** Turn on the AEB specific UART1 and call it /dev/ser1
  - **At:** CYGPKG_IO_SERIAL_ARM_AEB
  - **Change:**
    - from: `# user_value 0`
    - to: `user_value 1`

  - **At:** CYGPKG_IO_SERIAL_ARM_AEB_SERIAL1
  - **Change:**
    - from: `# user_value 0`
    - to: `user_value 1`

- Do this if you have the AEB-1C board

- **Purpose:** Change memory model to AEB-1C board (256K) rather than AEB-1B board (128K)
  - **At:** CYGHWR_HAL_ARM_AEB_REVISION
  - **Change:**
    - from: `# user_value B`
    - to: `user_value C`

- **Purpose:** Change the libc default console to our device defined above (this is for printf, etc)
  - **At:** CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE
  - **Change:**
    - from: `# user_value " \\/dev/ttydiag"`
    - to: `user_value " \\/dev/ser1"`

After this settings the libraries have to be built.

### 9.3.2 Build your sources

Then you can compile your program with this script or with a makefile and then running the arm-elf-objcopy commands afterwards.

```bash
## beginning of script called mybuild
## USAGE: mybuild file
## where file is prefix of source code file.c
```
## example: mybuild sample
rm -f $1.o
rm -f $1
rm -f $1tmp
rm -f $1tmp2
rm -f *.uue
arm-elf-gcc -mcpu=arm7di -c -o $1.o -Wall -I/d/ecos-work/aeb/config_nogdb_install/include -ffunction-sections -fdata-sections $1.c
arm-elf-gcc -mcpu=arm7di -L/d/ecos-work/aeb/config_nogdb_install/lib -Wl,-gc-sections -o $1 $1.o -Ttarget.ld -nostdlib
arm-elf-objcopy --strip-debug $1.exe $1tmp
arm-elf-objcopy -O binary $1tmp $1tmp2
echo Now you have to start uuencode and translate the binary $1tmp2...
echo A version for windows can be downloaded at http://home.swbell.net/ravntree/wuudoall.html
##uuencode $1tmp2 $1tmp2 | tr "" " " > $1arm
##uuencode not standard in cygwin
## end of script

Here is what is going on.

The first 5 lines are cleanup.
The first arm-elf-gcc compiles the source for your application.
- mcpu=arm7di  Picks the cpu for the AEB-1.
- c          Compile only, no linking.
- o $1.o     The object file name (sample.o).
- Wall       Report all warnings.
- I$HOME/ecos-work/install/include
Directory for the include files in the application source code:
#include <stdio.h> /* printf , etc. */
#include <cyg/kernel/kapi.h> /* All the kernel specific stuff */

-ffunction-sections
Removes unused functions from object file. Keeps size of executable down.

-fdata-sections
Removes unused data from object file. Keeps size of executable down.

$1.c  Source file, first argument in the call.

The next arm-elf-gcc links the source to the eCos library that was built.
- mcpu=arm7di  picks the cpu for the AEB-1 (maybe redundant, I don't know)
- L$HOME/ecos-work/install/lib
Where to go for the library
-Wl,-gc-sections Tells the linker to actually do the -ffunction-sections and -fdata-sections stuff, above.
- o $1  The output file name (sample).
$1.o    The object file to be linked.
- Ttarget.ld  The linker script that the eCos build creates.
-nostdlib Leave out the gcc libraries thus only using the eCos library (which nicely include a C-API, etc.)

The first arm-elf-objcopy strips the debug stuff (if any) from the linked code.
The linked file (sample)
The output of this first objcopy

The next arm-elf-objcopy "relocates" the code and turns it into executable code for the AEB.
The input.
The output

Since there is no uuencode included in the Cygwin environment you either have to download one from the gnu-sharutils or use a windows based application, which can be downloaded from http://home.swbell.net/ravntree/wudoall.html.

9.3.3 Edit the preferences in the windows based uuencoder
The following procedure needs to be performed:
1. Click on Preference->Encode. The Encode Preferences dialog will appear.
2. Click on the Write Encoding Table and check the box to disable it.
3. Click on the Write Size Record and check the box to disable it.
4. Click on the Use BackQuote For Spaces and check the box to disable it.
5. Click on the OK button.

The uuencode translates the AEB executable to an ASCII form that the AEB downloader will accept.

9.3.4 Download the uuencoded file
Start a Hyperterminal (Start- >Program- >Accessories) with the settings as in section 3.2.2 in the AEB User Guide (8 Data bits, 1 Stop bit, No parity and no flow control).

1. Press reset on the board, the following message will appear on the screen.
   ARM Evaluation Board Boot Monitor 0.01 (19 APR 1998)
   Press ENTER within 2 seconds to stop autoboot
2. Press reset within 2 seconds.
3. Type ‘download c000’.
4. Send the file with ‘Transmit- >Send Textfile’ from the menu in the Hyperterminal.
5. Type ‘go c040’.

In the Hyperterminal it will look something like this:

ARM Evaluation Board Boot Monitor 1.01 (18 NOV 1999)
Press ENTER within 2 seconds to stop autoboot
Boot: download c000
Ready to download. Use 'transmit' option on terminal emulator to download file.
Loaded file TESTPR~2 at address 0000c000, size = 45872
Boot: go c040
10 APENDIX C

10.1 MODIFICATIONS TO COMPILIE AXIS’ BLUETOOTH STACK

The files bluetooth.c and bluetooth.h integrate the stack with Linux so these files must be modified when porting to eCos. Also local.h has to be modified to correspond to eCos. Since the line #include “include/somefile.h” is present in several files one has to make a symbolic link in the src directory:

```
bash.exe-2.04$ ln -s ../include/linux/bluetooth include
```

When configuring eCos ‘TTY’, ‘TTY mode HAL/diag channel’ and ‘Standard input/output functions’ must be included. This will for instance enable printf, which is used in almost every file (the output will be printed in gdb’s console window when running arm-elf-gdb).

a) Since the application is supposed to run as an embedded program it is preferred not to dynamically allocate memory. In btmem.c is now the buffer statically allocated.

There are some places where malloc is used. The functions kmalloc and kfree have been redefined in btcommon.h to bt_pool_malloc and bt_pool_free.

```
#ifdef _ECOS
  //memory
  #define kmalloc(size,prio) bt_pool_malloc((size))
  #define kfree(obj) bt_pool_free((obj))
  //sync
  #define wake_up_interruptible(p) cyg_semaphore_post(p)
  #define interruptible_sleep_on(p) cyg_semaphore_wait(p)
  #define sleep(p) /*nothing*/
#endif
```

These functions use memory from a statically allocated memory pool. The functions are located in bluetooth.c.

Miscellaneous changes:
1. In hci.c: insert_cmd. A check has been added if kmalloc fails.

2. In sdp.c: process_service_search_req. A check has been added if kmalloc fails.

In bluetooth.c a new define has been added, MAX_APPL_DATA_SIZE, this restricts the maximum number of bytes sent from the application to rfcomm.

b) The wait-queues have been declared as semaphores. This means that the wake_up_interruptible and interruptible_sleep_on functions
have been redeclared in `btcommon.h` as `cyg_semaphore_post` and `cyg_semaphore_wait`.
In `hic.c`;
`cyg_semaphore_init(&hci_wq,0);`

c) Timers in `hci.c` have been replaced by eCos-timers.

### 10.2 MODIFICATIONS TO eCos SOURCES

1. To notify a thread when data arrives on `/dev/ser0` a semaphore has been added. The changes are:
   In `Red Hat\eCos\packages\io\serial\v1_3_1\include\serialio.h`:
   ```c
   cyg_sem_t Data_Rec_Sem;
   ```
   In `Red Hat\eCos\packages\io\serial\v1_3_1\src\arm\aeb_serialio.c`:
   ```c
   cyg_semaphore_init(&Data_Rec_Sem,0);
   ```
   In function `aeb_serial_init`:
   ```c
   cyg_semaphore_post(&Data_Rec_Sem);
   ```
   This means that the value of the `Data_Rec_Sem` equals the number of bytes received.
   The receiver process looks like:
   ```c
   void receiver(cyg_addrword_t data)
   {
       static cyg_count32 sem_count;
       Cyg_ErrNo err;
       cyg_uint8 buf[CYGNUM_IO_SERIAL_ARM_AEB_SERIAL0_BUFSIZE];
       //the bufsize is configurable in eCos config-program
       cyg_io_handle_t ser0_hdl;
       err = cyg_io_lookup("/dev/ser0", &ser0_hdl);
       if(ENOERR != err){
           printf("Couldn't find /dev/ser0. Killing the process!!!!!!!!!\n");
           cyg_thread_exit();
       }
       sem_count=0;
       printf("Receiver started\n");
       while(1) {
           cyg_semaphore_wait(&Data_Rec_Sem);  //blocks here if no data
           available on UART0
           sem_count=1;
           while(cyg_semaphore_trywait(&Data_Rec_Sem)){sem_count++;}
           err = cyg_io_read(ser0_hdl, &buf, &sem_count);
           if(ENOERR == err)
               //send the data to HCI
               bt_receive_lower_stack(buf,sem_count);
           }
       }
   }
   ```

2. To be able to set the baud rate of UART0 to 57600 the baud divisor 26 was added in the `select_baud` array in `Red Hat\eCos\packages\io\serial\v1_3_1\src\arm\aeb_serialio.h`
## 11 APPENDIX D

### 11.1 EXPANDING AEB-1’S MEMORY

Two new SRAM were piggybacked on top of the existing ones. Components needed:

(2) 512K x 8 SRAM. select a speed that is reasonable for your needs and budget. The Toshiba TC554001FL-70 or Samsung K6T400C1B will do fine.

These 2 RAM chips are to be mounted piggy-back style on top of the two SRAM IC's that are already on the AEB-1. All of the leads of the new RAM IC's (with the exception of pins 1, 2, 22, and 30) are to be soldered to the corresponding pins of the SRAM IC's they are mounted on.

This will connect all of the Data bus (D0--D15), most of the Address bus (A1--A16), plus /OE, /WR, Power and GND. This leaves only 8 wires that need to be manually connected to complete the expansion. Here is a list of the connections to be made:

<table>
<thead>
<tr>
<th>Signal</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address A17</td>
<td>JP1 pin 26</td>
<td>pin 2 of added RAM chips</td>
</tr>
<tr>
<td>Address A18</td>
<td>JP1 pin 27</td>
<td>pin 30 of added RAM chips</td>
</tr>
<tr>
<td>Address A19</td>
<td>JP1 pin 28</td>
<td>pin 1 of added RAM chips</td>
</tr>
<tr>
<td>Chip Select 2</td>
<td>JP2 pin 23</td>
<td>pin 22 of added RAM on U4</td>
</tr>
<tr>
<td>Chip Select 3</td>
<td>JP2 pin 24</td>
<td>pin 22 of added RAM on U3</td>
</tr>
</tbody>
</table>

A total of 8 wire segments.

**Memory Configuration and testing:**

The memory interface logic in the Sharp LH77790 must be configured for this new memory. It is also a good idea to test the memory to be sure that all connections are OK.

Here is a small code sample that does this, written by Neil Puff at Green Hills Software:

```c
#define START3  ((volatile int *)0xFFFFA00C)
#define STOP3   ((volatile int *)0xFFFFA02C)
#define SDR3    ((volatile int *)0xFFFFA04C)
#define BCR3    ((volatile int *)0xFFFFA10C)
#define XMSTART 0x48000
#define XMSTOP  0xC8000
#define SEED    0xC0FFEE42   /* don't set this to 0x0000 !! */

int count;
int test;

*START3 = XMSTART;
*STOP3  = XMSTOP;
*SDR3  =  0x7C08;  /* set for maximum wait */
*BCR3  =  0xF090;

/* and now for a (ahem) quick memory test ....
* This test will write a pseudorandom sequence into all of
```

63
/* the expanded memory, and then go back and make sure that
 * the sequence is correct.
 */
test = SEED;
for(count = XMSTART; count < XMSTOP ; count += 4) {
    *((volatile int *) (count)) = test;
    if(test)
        test = ((test << 1) + (((test & 0x80000000) >>31)^((test & 0x10000000)>> 28));
    else
        test = SEED;
#ifdef EMA_SHOW
    if(!(count % 0x10000))
        printf("Count = %X (write pattern)\n",count);
#endif
}
/* check the pattern we just wrote */
test = SEED;
for(count = XMSTART; count < XMSTOP ; count += 4) {
    if(*((volatile int *) (count)) != test) {
        printf("memory error! Pattern = %X at location %X\n",test, count);
        count = XMSTOP;
        break;
    }
    if(test)
        test = ((test << 1) + (((test & 0x80000000) >>31)^((test & 0x10000000)>> 28));
    else
        test = SEED;
#ifdef EMA_SHOW
    if(!(count % 0x10000))
        printf("Count = %X (read pattern)\n",count);
#endif
}

11.2 USING eCos WITH THE NEW MEMORY

First of all a new GDB-stub has to be flashed into ROM on AEB. This is done exactly as described in the documentation, see “ARM AEB-1 Hardware Setup 43”. Just remember that when sending the file to the board when using HyperTerminal use ‘transmit->send text file’.

A standard stub should be no problem building, but it was, you should just open the eCos configuration tool and by selecting ‘build->template->AEB-1 and ‘stubs’ ’ and then change from revision B to revision C under the HAL-package.

The file ml_arm_aebC_rom.ldi had to be changed from:

```c
SECTIONS
{
    SECTIONS_BEGIN
    SECTION_rom_vectors (rom, 0x4018000, LMA_EQ_VMA)
    SECTION_text (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fini (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata1 (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fixup (rom, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_gcc_except_table (rom, ALIGN (0x4), LMA_EQ_VMA)
    __reserved_vectors = 0; . = __reserved_vectors + 0x1000;
    SECTION_data (ram, ALIGN (0x4), FOLLOWING (.gcc_except_table))
```
WOSP - Wireless Open Source Platform

SECTION_bss (ram, ALIGN (0x4), LMA_EQ_VMA)
   __reserved_not_mapped = ALIGN (0x4); . = __reserved_not_mapped + 0x7000;
SECTIONS_END
}

to

SECTIONS
{
SECTIONS_BEGIN
SECTION_rom_vectors (rom, 0x4018000, LMA_EQ_VMA)
SECTION_text (rom, ALIGN (0x4), LMA_EQ_VMA)
SECTION_fini (rom, ALIGN (0x4), LMA_EQ_VMA)
SECTION_rodata (rom, ALIGN (0x4), LMA_EQ_VMA)
SECTION_rodata1 (rom, ALIGN (0x4), LMA_EQ_VMA)
SECTION_fixup (rom, ALIGN (0x4), LMA_EQ_VMA)
SECTION_gcc_except_table (rom, ALIGN (0x4), LMA_EQ_VMA)
   __reserved_vectors = 0; . = __reserved_vectors + 0x1000;
   __reserved_not_mapped = 0x1000; . = __reserved_not_mapped + 0x7000;
SECTION_data (ram, ALIGN (0x4), FOLLOWING (.gcc_except_table))
SECTION_bss (ram, ALIGN (0x4), LMA_EQ_VMA)
SECTIONS_END
}

The matching changes have to be done in mlt_arm_aebC_rom.mlt just to see
the changes in the configuration tool’s memory layout tool.
And then every occurrence of the RAM size 4800 in the files
mlt_ram_aebC_rom.h, mlt_arm_aebC_rom.ldi and
mlt_arm_aebC_rom.mlt
needs to be replaced with C8000.
Now the processor has to be set up to use this new memory. The stub does this
with a macro named PLATFORM_SETUP1 located in hal_platform_setup.h
this file will look like this after the changes:

#define AEB_SRAM    .long
0xFFFFA008,0x00008000,0x00048000,0x00007804//BANK2
#define AEB_SRAM2   .long
0xFFFFA0OC,0x00048000,0x000C8000,0x00007808//BANK3
#define AEB_BAD     .long
0xFFFFA010,0x000C8000,0x01000000,0x00000000/BANK4
#define BCR3        .long
0xFFFFA10C,0x0000F090 //new 512KB SRAM, //16bit,max wait-state,CE2 and CE3

#ifdef CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS
#define PLATFORM_SETUP1                                                   
   ldr r1,=CYG_DEVICE_CCR                                          ; 
   mov r2,#CCR_I                                                          ; 
   strb r2,[r1,#0]      /* invalidate... */                                   ; 
   mov r2,#0                                                             ; 
   strb r2,[r1,#0]      /* and disable the cache. */                  ; 
   ldr r1,=segment_register_setups                               ; 
   str r3,[r1,#0x00]                                                        ; 
   ldr r3,[r1,#12]  /* segment address */                        ; 
   cmp r2,#0                                                           ; 
   beq 20f                                                             ; 
   ldr r3,[r2,#0]  /* segment start */                        ; 
   str r3,[r2,#0x00]                                                        ; 
   ldr r3,[r1,#8]  /* segment end */                         ; 
   str r3,[r2,#0x20]                                                     ; 
   ldr r3,[r1,#12]  /* segment flags */                        ; 
   str r3,[r2,#0x40]                                                   ;
#endif
add r1, r1, #16 /* next segment */
add r1, r1, #4 /* next segment */
20: ldr r1, =new_bcr_register_setups
30: ldr r2, [r1, #0] /* bcr address */
cmp r2, #0
beq 40f
ldr r3, [r1, #4] /* bcr value */
str r3, [r2, #0x00]
add r1, r1, #4 /* next segment */

segment_register_setups:
AEB_SRAM /* segment 2 */
AEB_SRAM2 /* segment 3 */
AEB_BAD /* segment 4 */
.long 0
new_bcr_register_setups:
BCR3 /* segment 2 */
.long 0

40:
#else
#define PLATFORM_SETUP1
#endif

11.3 BUILDING THE STUB UNDER WINDOWS

When building the stub there will be a couple of errors when the stub’s checksum is calculated by a tcl-script named flash_chksum.tcl this script needs to be copied from /program files/Red Hat/eCos/packages/hal/arm/aeb/v1_3_1/src to "location of your configuration"/"name of your conf"_build/hal/arm/aeb/v1_3_1/src and the makefile located in v1_3_1 needs to be modified from:

# Then build version with checksum from previousl y built image.
$(CC) -c -DCALLCHECKSUM=` $(dir $<) flash_cksum.tcl …
to
# Then build version with checksum from previously built image.
$(CC) -c -DCALLCHECKSUM=`src/flash_cksum.tcl …

Now will the compilation run until the binary should be uuencoded. Since there is no such tool in Cygwin this will fail. Instead a windows based uuencoder can be used such as that one mentioned above.
When this is done the only thing left is to download the new stub to the board with the commands.
download c000
flashwrite 4018000 c000 8000
plugin eCos

11.4 USING THE NEW STUB

A new configuration with the same changes as above but now performed in the mlt_arm_aebC_ram files needs to be compiled and then as usual linked with your sources in the makefile.
12 APPENDIX E

12.1 INTERRUPT MODEL IN eCos

Quoted from eCos’s documentation.

“Interrupt handlers are actually a pair of functions, one of which (the interrupt service routine, or ISR) is executed immediately and runs with that interrupt disabled. Since interrupts are disabled for the duration of the ISR, the ISR should be very brief and should not use any system services.

After the ISR exits, but before the kernel scheduler is invoked again, a delayed service routine (DSR) will be invoked. It executes with scheduling disabled, but with interrupts enabled, so that further invocations of the same DSR can be queued. The DSR can use some producer-side system calls, but it should be carefully crafted to avoid using any call that might put its thread to sleep. One of the few examples of safe calls is cyg_semaphore_post(); the non-blocking versions of some system calls are also safe. A call that is unsafe is cyg_mutex_lock(), since it will block if the mutex is already locked by another thread.

Finally, eCos has a formalism for installing low level handlers which bypass the kernel mechanisms described above. A program can install a vector service routine (VSR), which will be invoked instead of the kernel’s usual exception or interrupt handling. The VSR will typically be written in assembly language.

VSRs are associated with vector numbers in the exception space, just like exception handlers (although there are some variations -- architectures in which there are no exceptions in the eCos sense). The main difference between VSRs and exception handlers is that VSRs bypass the kernel’s usual mechanisms.”

As a conclusion any can say.
ISRs are called right now regardless of kernel state, therefore you cannot do calls into the kernel from an ISR.
DSRs are called (when requested) as soon as possible after the associated ISR, when the kernel state is such that it’s safe to call into the kernel.
A DSR still cannot sleep, because it is not a thread context; only threads can sleep. Therefore only operations, which cannot block may be called from a DSR.
13 APPENDIX F

13.1 MODIFICATIONS TO COMPILE THE WAP STACK

A wrapper (wrapper.h wrapper.c) was created to avoid changing every call to
pthread-functions from the WAP-stack. This wrapper-layer redefines the
pthread-features to thread-functions defined by eCos.

A variable size memory pool has been added. Every call to malloc in the
WAP-stack uses this pool. The functions on the memory pool and the size
have been specified in wrapper.h, the size is defined by
WAP_MALLOC_SIZE. The initialisation is done from cyg_user_start() with
the call to wap_malloc_mem_init().

CREATE_OBJECT has been redefined and a new FREE_OBJECT macro
added. By using this macros there should be no mistakes done.

In utils.h
#ifdef __ECOS //use a mempool
#define CREATE_OBJECT(object,object_type)  
   (object)=(object_type*) wap_malloc(sizeof(object_type)); 
   if ((object)==NULL)  
      errno_abort("create_object()")

#define free(object)  wap_free(object)
#define FREE_OBJECT(object) wap_free(object)
#else

NOTE: When adding more layers above WDP replace free(addr) with
FREE_OBJECT(addr)

In event.c a new initialisation function ‘void Event_init(void) ‘has been added,
this function initialises the mutex-variables and the condition-variables used in
the event-class. This function is called from wdp_machine.c in function
WDPMachine_run.

As explained in the porting section of Bluetooth above when sending the
T_DUnitdata structure directly to Bluetooth the TptAddr structure needs the
__attribute__((packed)) command.