



A Complete Real-time Operating System Environment for Embedded System Applications.

Contents

1	Introduction.....	7
1.1	Purpose of this document	8
1.2	Concepts	8
1.3	Razor architecture and functionality	8
1.4	Who should read this document	8
1.5	Some basic concepts	8
1.6	A bird's-eye view of the Razor development, test and operational environment	9
1.6.1	Razor platform - hardware architecture	9
1.6.2	Razor platform applications-oriented hardware	11
1.6.3	Razor platform - software architecture.....	12
1.6.4	How has isolation been achieved and why is it so important?	12
1.6.5	The Pboot approach	14
1.6.6	The significance and advantages of the Pboot approach	17
2	A detailed survey of Razor's components.....	18
2.1	Introduction	18
2.2	Descriptions - Major Razor Components	20
2.2.1	PBOOT	20
2.2.2	Generating A Razor Program Unit	31
2.2.3	Memory Management	34
2.2.4	Restart/Recovery	45

2.2.5	Program Handling	49
2.2.6	The Razor Development & Testing Environment.....	56
2.2.7	Historical background	57
3	Glossary	61
4	Ownership rights and Interpretation.....	66
5	APPENDIX	68
5.1	The world of real-time	69
5.1.1	Some important real-time processing concepts.....	70
5.2	Embedded systems.....	73
5.2.1	Speculating as to the future of real-time embedded systems.....	74

Copyright

This document was published 2000-09-01 in Sweden by Ericsson Utvecklings AB (UAB).

All rights are reserved. The contents of this document are the property of the Ericsson Corporation and are subject to local and international copyright law. Any reproduction in whole or in part is strictly prohibited without the expressed written consent of the publisher.

For further information please contact Ingrid.Wahlborn@uab.ericsson.se

Trademarks

OSE and OSE DELTA are registered trademarks of ENEA DATA AB.

All other terms marked ® are trademarks of Ericsson unless specified otherwise.

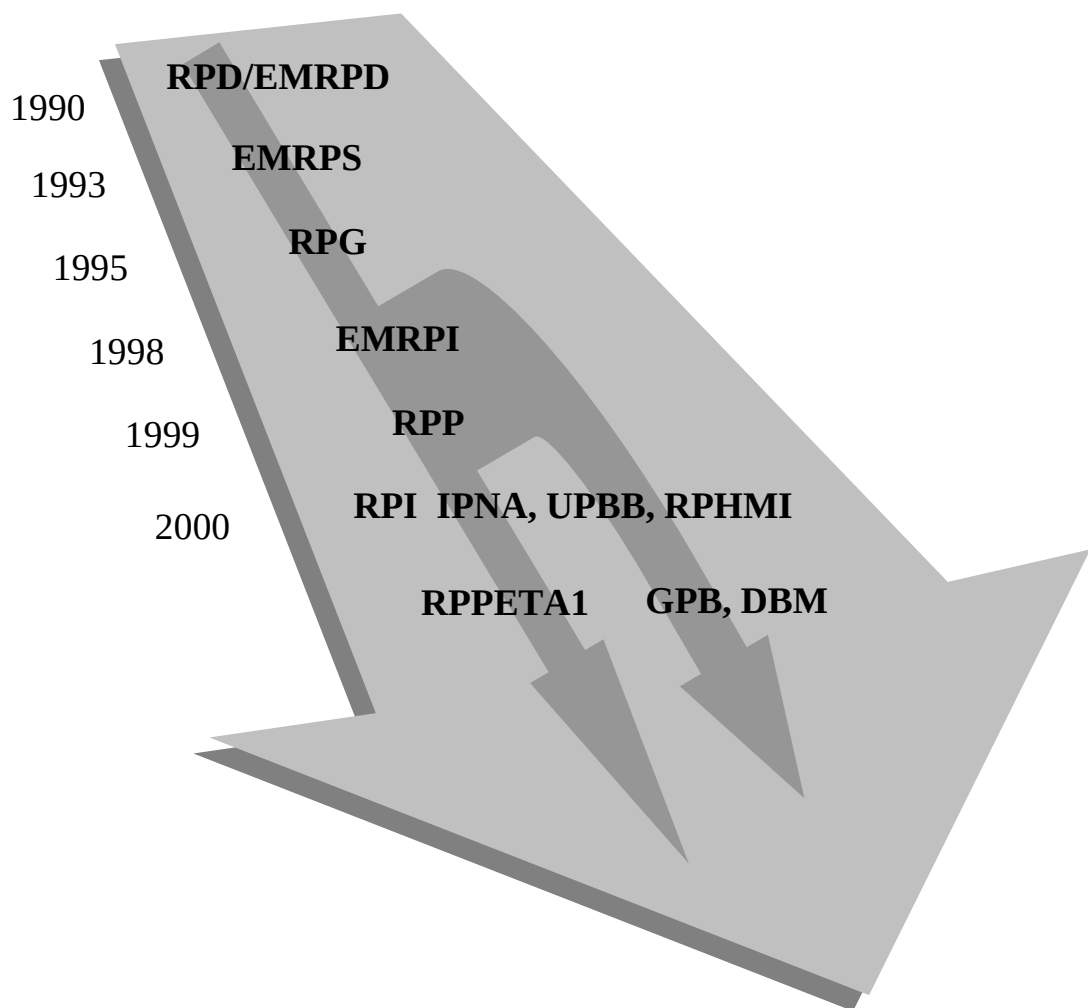
This page is intentionally left blank

Foreword

Embedded real-time applications are becoming increasingly complex - the amount of time available for the development of these products is continually decreasing while consumers and customers are constantly asking for more – more functionality and more ease-of-use features. High-performance microprocessors are beginning to appear more and more in an ever increasing variety of products. The range of products is awesome, covering telecommunication switches, cellular phones, cameras, industrial robots, medical instruments, home appliances, computers and military systems. The list goes on and on.

This is the reality faced by embedded real-time software/hardware development organizations and they need all the help they can get. The use of a well-integrated development environment that includes a feature-rich Real-Time Operating System along with a toolbox containing a wide selection of auxiliary routines is key to effectively meeting both present and future challenges. To this end, UAB offers the Razor real-time operating environment to aid the embedded real-time applications developer in achieving goals such as on-time delivery and the widespread market acceptance of Ericsson products.

Razor's track record

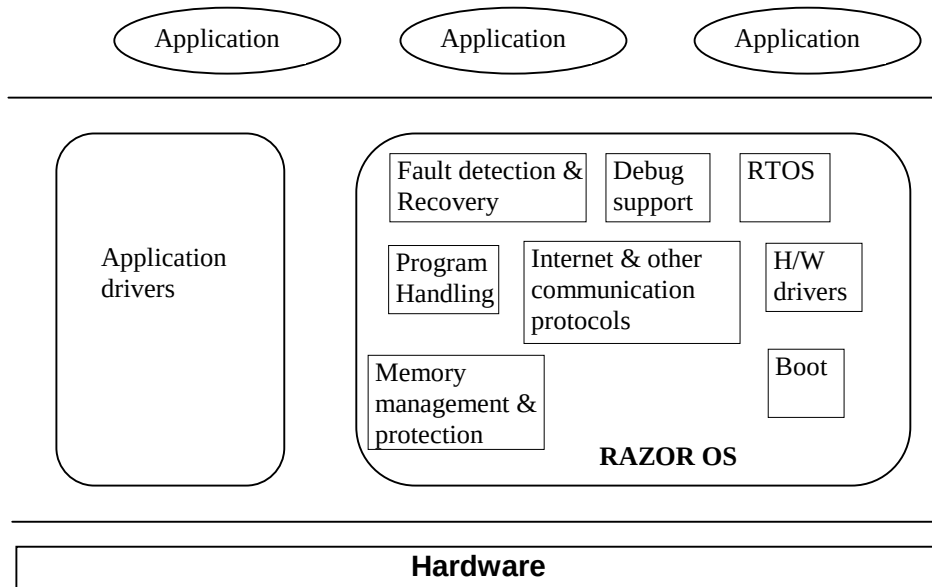


This page is intentionally left blank

1 Introduction

Razor is a real-time operating environment that has been specifically designed to satisfy a number of essential embedded-system characteristics.

The figure below provides a simplified sketch of the operating environment of the Razor real-time system.



It is important to note the high degree of isolation that has been attained between the OS and the application environment. System application engineers need not involve themselves in the hardware or software aspects of the various elements included in the OS platform, they need only concentrate on the software and hardware aspects of their applications. In particular, please note that application drivers, i.e., software drivers that initiate and control the execution of the various hardware-oriented application functions, are not contained within the operating system kernel. This makes for very good isolation between the kernel and applications hardware-oriented functions. The advantages of this architecture will be penetrated in greater detail in section 1.6.3, Razor platform - software architecture.

Aside from offering high availability, high performance and reliability, a professional real-time platform must also provide a powerful debug and test environment that is accessible and that adequately fills the needs of both OS system engineers and applications engineers.

Razor is the result of over 10 years of real-time embedded systems development.

1.1 Purpose of this document

This document should provide the reader with a working knowledge and a reasonable understanding of the Razor operating environment and its application in real-time embedded systems.

1.2 Concepts

The document reviews some of the basic concepts related to real-time processing as well as some contemporary ideas surrounding embedded systems.

1.3 Razor architecture and functionality

The document explores the architecture of the Razor environment and outlines the functions of the various components that are included.

1.4 Who should read this document

The primary target group for this document is system designers who require an introduction to the Razor operating environment and its application in the realm of modern real-time embedded systems.

Hopefully, the document will also appeal to those having a more general interest in the subject of real-time embedded application environments including marketing and sales personnel who will be marketing Razor-based systems.

1.5 Some basic concepts

The appendix at the end of this document reviews a number of fundamental concepts related to the Razor platform. Included are:

- A definition of the term real-time and some important real-time concepts.
- A definition of the term embedded system and speculations as to the future of such systems.

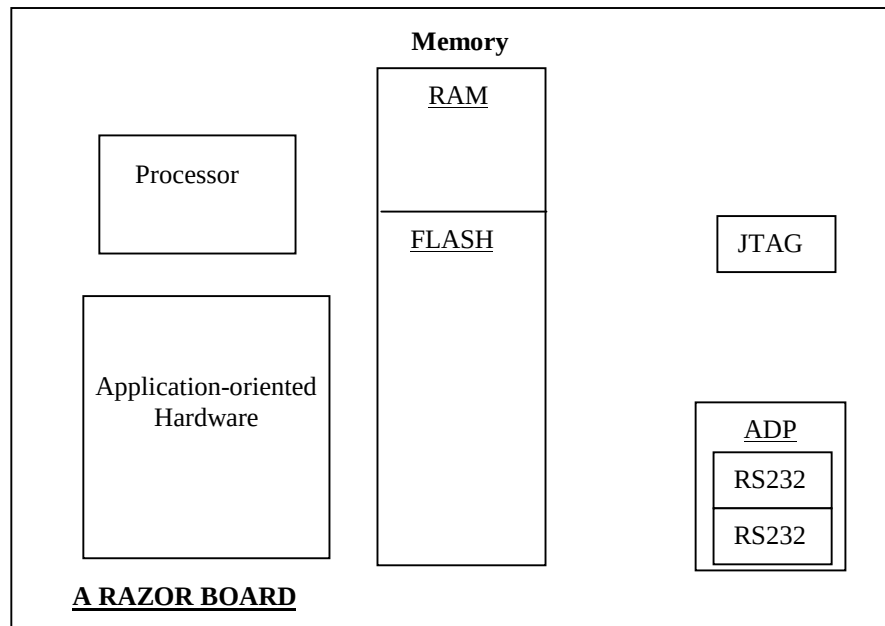
It is suggested that readers who are not entirely familiar with these areas first read the appendix before continuing.

1.6 A bird's-eye view of the Razor development, test and operational environment

1.6.1 Razor platform - hardware architecture

Please refer to document UAB/D-99:232 Uen, Generic Requirements on H/W for a Razor Platform.

The general layout and typical component content of a Razor board is illustrated in the figure below:



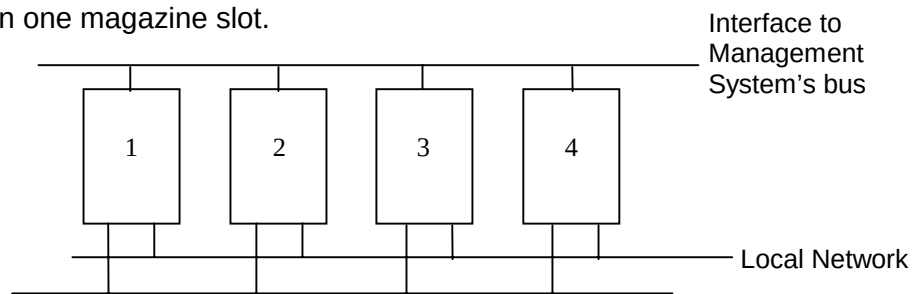
The following components represent the basic architectural requirements of a Razor applications environment:

- An ADP (Advanced Debugger Port) including one RS232 port.
- A Power PC.
- RAM memory, preferably with parity checking.
- FLASH memory.
- Watchdog. This feature may be disregarded at the discretion of the applications architect. Note that fault tolerance will diminish considerably if the feature is not included.

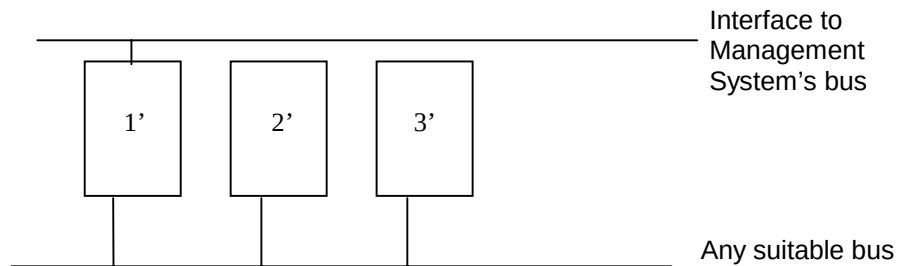
The following components represent optional features that may be added to the basic Razor applications environment:

- Ethernet for host communication. Optionally, may be used for intra-board communication over a "local network". Dual Ethernets may be required for redundancy purposes.
- A second RS232 port.
- A management system. (Note that a Razor environment is capable of autonomous operation.)

1. The following illustrates two examples of Razor applications that span more than one magazine slot.



The situation shown above depicts 4 boards mounted in one magazine, all containing a CPU, RAM and Flash memories. Boards 1, 2 and 3 have each been detailed a portion of the entire application and each communicates directly with the Management System via a communication link that has been configured. The boards also communicate with one another via the “Local Network” using the OSE Link Handler protocol. Board 4 in this example is an online spare that will assume the load in the event that boards 1, 2 or 3 should fail.



The situation depicted in the second diagram above shows 3 boards mounted in one magazine where only one (board 1') is configured with a CPU, RAM and Flash memories. Boards 2' and 3' simply contain hardware that supports the applications' need for devices, etc. Only board 1' has contact with the Management System and communicates with the other boards via the bus structure selected by the application system architect.

1.6.2 Razor platform applications-oriented hardware

The following standard hardware facilities are supported by the Razor environment.

PCI bus – (Peripheral Component Interconnect) an interconnection system between a microprocessor and attached devices.

CPCI bus – Compact PCI bus defines standard card and contact configurations, used in some special cases.

Ethernet – Razor supports both 10 and 100 Mbit/s speeds.

Rapid I/O – Not available as yet. A possible future replacement for the PCI bus.

E1/T1 interface – Razor supports European/USA transmission speeds (2 and 1.5 Mb/sec).

STM1 interface – Requires application support.

ATM – Requires application support.

A number of proprietary device buses facilitate the support of a diverse range of applications hardware, such as:

RPBSIC – An AXE serial RP bus interface.

RPBIC – An AXE parallel RP bus interface.

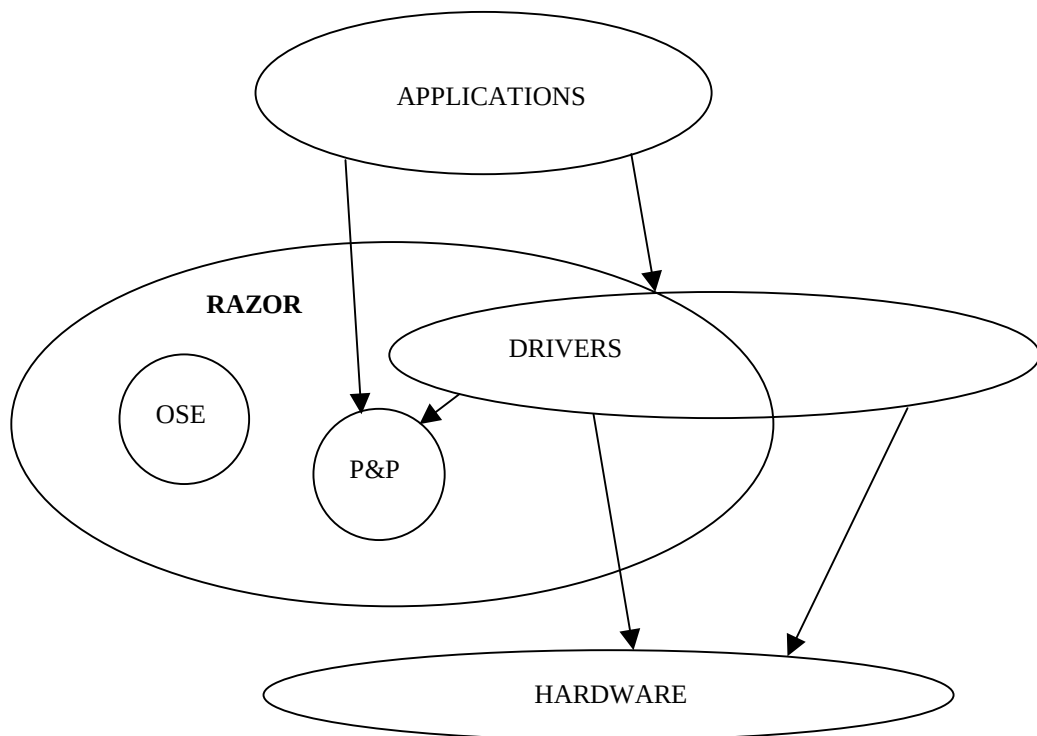
DSP clusters – Requires application support.

1.6.3 Razor platform - software architecture

As in the case of any other complex software product, the elegance of Razor is not easily illustrated via the use of a single simplified diagram - the system is composed of a number of interworking components that interface with one another in a more or less complex fashion.

Having said this, there are however two aspects of Razor that are fundamental and easily illustrated. They are, the isolation that has been attained between the operating environment and the applications environment and the degree of so-called "plug-and-play" that has been introduced.

The diagram below illustrates the relationship between the applications, the Razor components and the various hardware drivers. It is just this structure that makes it relatively easy to apply this environment to a wide range of applications. In particular, note the position of the applications in relation to the drivers that they make use of. Also note that certain drivers are included in the Razor environment and that others are constructed specifically for the particular application by the applications design staff.



1.6.4 How has isolation been achieved and why is it so important?

The level of isolation that has been attained is the result of a deliberate system design approach, i.e., a design that contributes to the separation of application-dependent software from application-independent software. The fact that this approach has also proven itself to be favorable in supporting the development process is an added benefit, i.e., applications and application-dependent drivers may be coded in parallel by the applications design staff thereby reducing the total development time required.

Application programs register their need for hardware drivers with the P&P (Plug & Play) component in Razor - the various drivers (both those included in the standard platform and those written for the specific application) register their existence with the same P&P component. Since this "registration" process goes on during system startup and not as part of the system generation process, a satisfactory level of Plug&Play is attained. Once the registration process is completed, the specific applications are cleared to make use of the drivers that they require.

The above translates into a number of advantages:

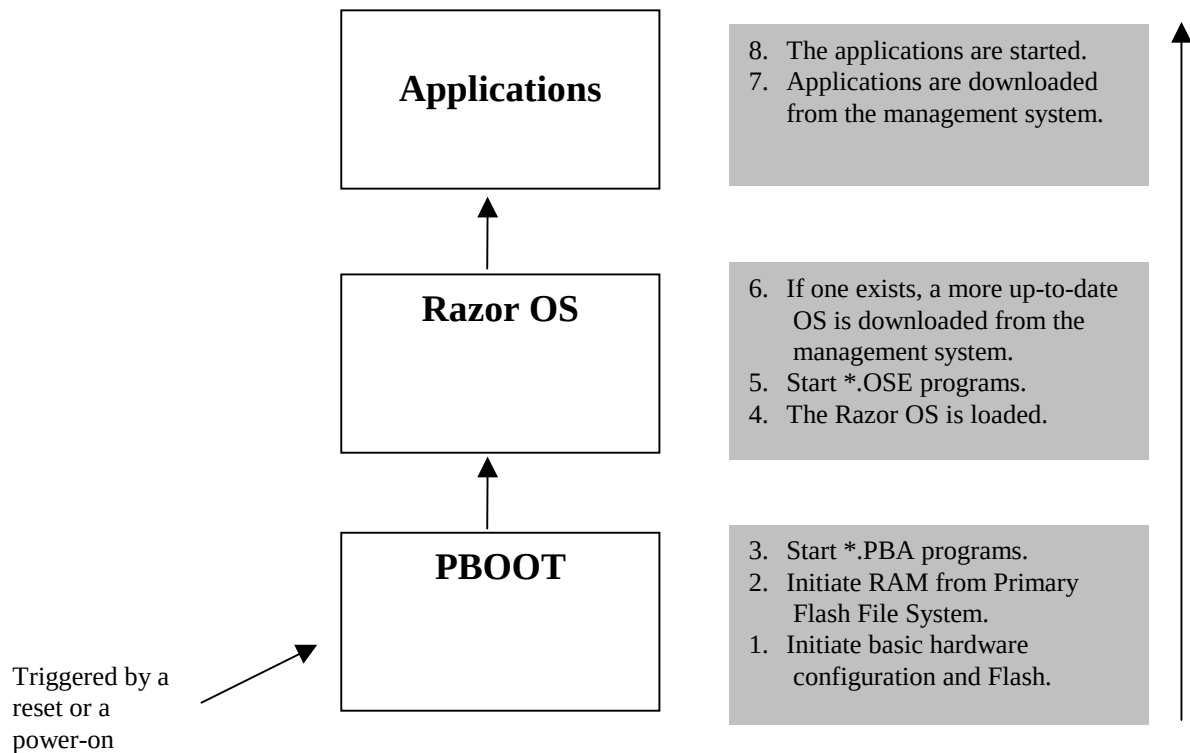
- The OSE Kernel requires no porting if the same processor configuration is used as in an earlier design.
- The P&P component may be used as is - conversion is, for all practical purposes, never required.
- New application programs must naturally be written for the new board. However due to the test facilities provided by the Razor platform, the testing of specific application-oriented hardware functionality may start long before applications are ready for testing.
- Driver services for a particular configuration may be setup at system initiation time and need not be "mapped in" at system generation time.

From Razor's point of view, both applications and drivers are just "ordinary" real-time programs that occasionally require its attention.

1.6.5 The Pboot approach

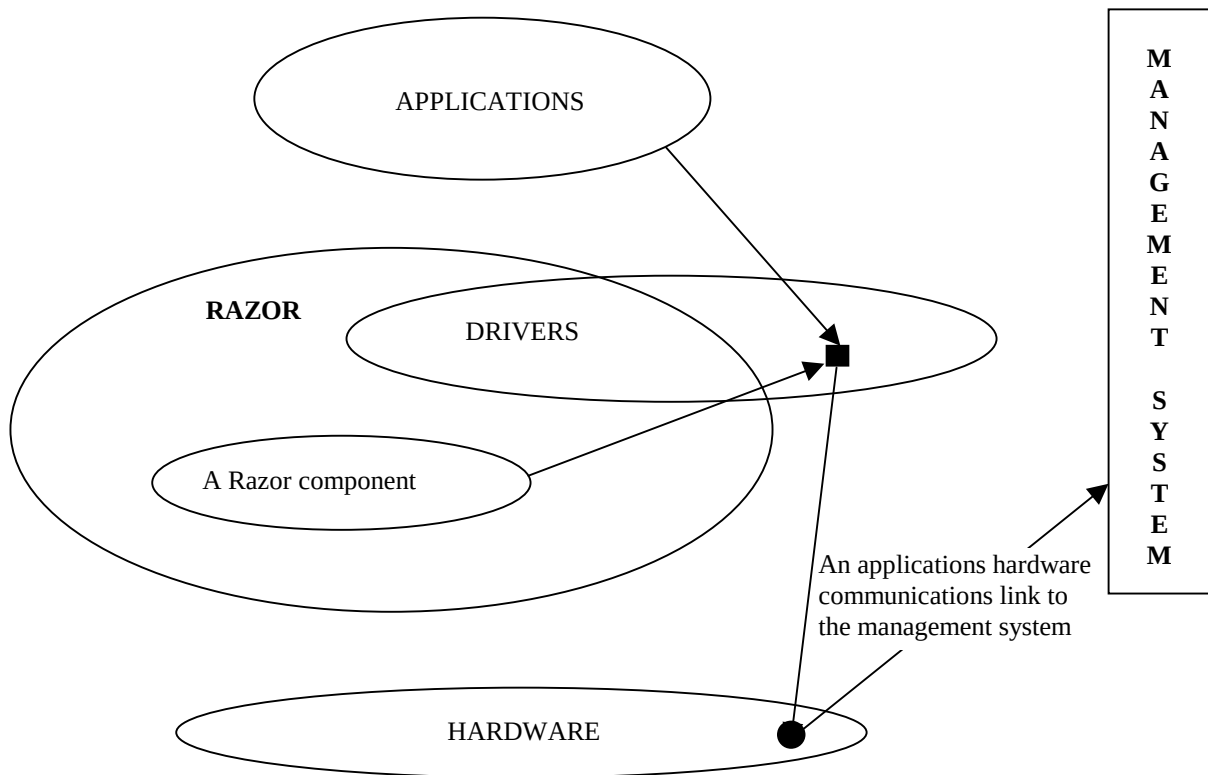
See section 2.2.1, PBOOT, for a more detailed description of the Pboot process.

The bootstrap routine has been especially designed to further enhance the platform's operational and test facilities.



An example of a *.PBA program:

Programs that are run in conjunction with Pboot. Such programs may either test certain application-sensitive hardware or setup specific hardware such as FPGAs (Field Programmable Gate Arrays). These programs are run in sequence, i.e., Pboot starts up one program and waits for it to terminate before starting the next PBA program. If many PBA programs are to be run, there is no guarantee as to the sequence in which they will be run.

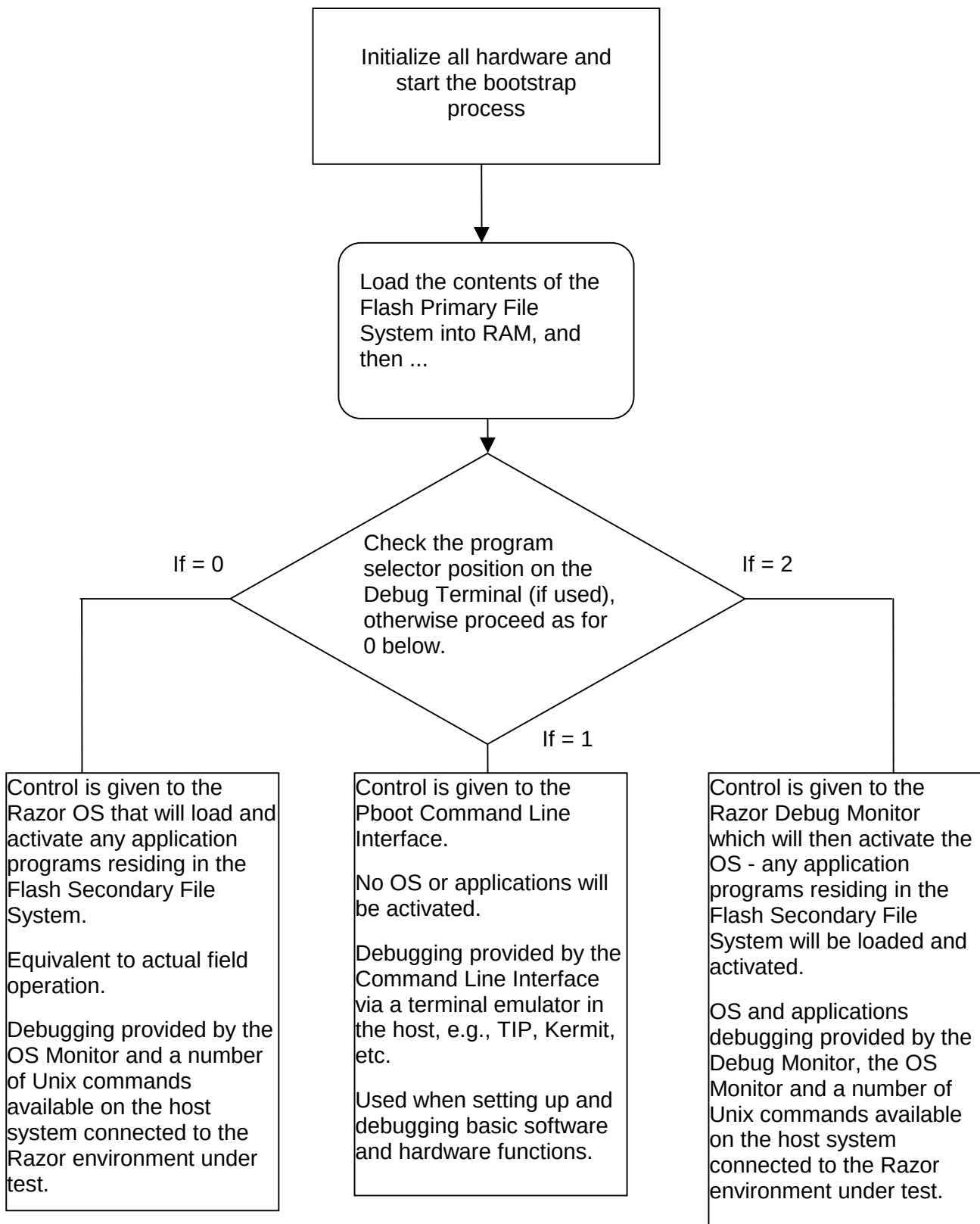


An example of an *.OSE program:

A program that runs immediately following the activation of the Kernel. These are generally special setup routines such as the example shown above.

Here, an *.OSE application program is run that sets up and activates a common communications channel that may be used by both the application and the components of Razor. In this particular case, the Razor platform does not have its own path (i.e., its own hardware and driver) to the management system and the application provides the hardware and driver required to access the management system.

The illustration below provides a highly simplified picture of the Razor bootstrap process, Pboot, and the various options available for debugging.



1.6.6 The significance and advantages of the Pboot approach

Development times are shortened through early debugging of both OS and hardware elements.

Aside from just initiating the hardware and software environment in preparation for loading the Operating Environment, Pboot also supports a highly versatile debugging environment for both system and applications programmers.

Once Pboot is initiated, the user may elect to proceed all the way to an operational system in which both the OS and the applications are initiated, or to an intermediate state, i.e., a system state that is suitable to the task at hand.

A programmer may wish to test the functionality of certain application-oriented hardware in order to specify the behavior of a software module that hasn't as yet been written. This may be easily done using Razor and specially written hardware test programs that run under Pboot.

Another programmer may wish to initiate and then test the functionality of an FPGA device (a Field Programmable Gate Array) long before any application software is ready for testing. Again, this may be done using Razor and specially written programs that run under Pboot.

A third programmer may wish to test certain new facilities in the operating environment that are to be implemented for a special project. Pboot may be primed so that it loads the operating environment whereupon the programmer can then test the new functionality.

2 A detailed survey of Razor's components

2.1 Introduction

The table below describes the contents of the Razor software environment and includes the major software components.

<u>Razor OS components</u>	<u>Description</u>
Aloader	Auto-loader for *.ose and *.auto programs stored in the PBOOT file system.
Auto DDS + stoptarget	Auto configuration of system debugging using PBOOT parameters. Errorcheck level and more.
BSP Man	Simple "plug and play" system for device drivers and applications conforming to BSP standards. Manages device drivers and applications.
AXE Communications	AXE - RP communication
Ethernet Inh	Ethernet Link Handler for the "local network".
Extend pool	Dynamic Signal Pool Extension. Avoids the need for pre-configuration of pool size.
Flash FS	Flash File System for program cache.
GSI	Easy to use AXE Group Switch Interface for low-performance E1/T1 access.
L_file	Local file utility functions to access files embedded in RPDOOT programs.
Loader	Program Loader (a function included in Program Handling)
Lomas	CPU Load Measurement
MMU	Memory Management (H/W and S/W)
Nvparam	Provides a signal handling Service for application access to Flash parameters.
OS monitor	The OS monitor component is a command-line monitor implemented in the operating system. Two types of users use this function - monitor applications and ordinary consoles.
OSE Delta	OSE Delta kernel, a real-time operating system provided by ENEA OSE Systems.
OSE Dump	OSE Delta post mortem analysis.
PBOOT	Primary BOOT, is a small micro boot and the first program to Execute after power-up, watchdog or other hard resets of the CPU.
PCI	PCI management.

Periodic test	Background H/W Testing
RDS	OS monitor interface to system debugger.
Dump	OSE kernel error handler (restart dump).
ROM-mon.	Debug Monitor (also referred to as ROM Monitor)
RTC	Real-Time Clock
V24 Inh	ADP V.24 Link Handler.
Watchdog	Watchdog support

Typical languages supported by the Razor environment are C, C++ and JAVA.

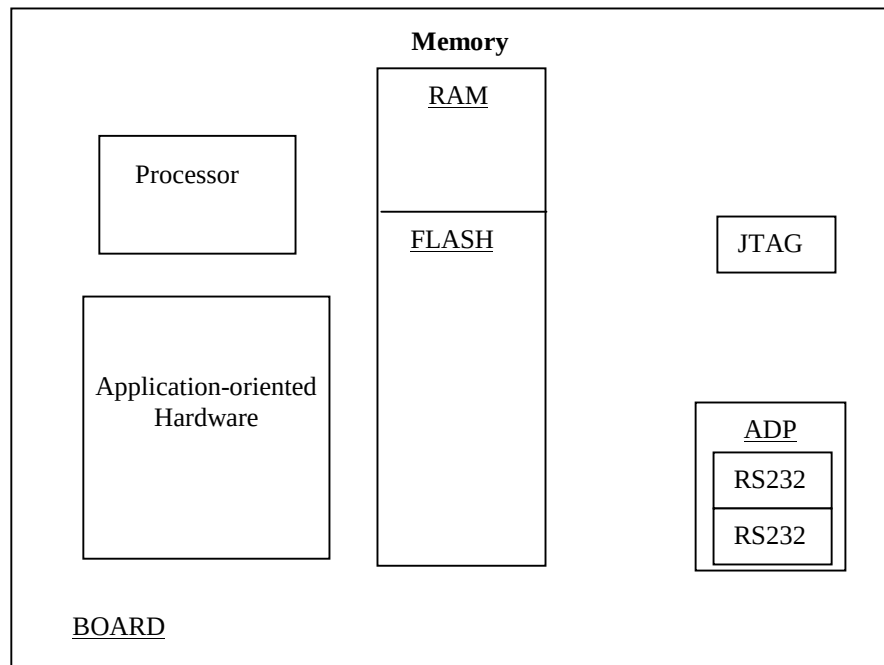
Note that TCP/IP support exists for FTP, TFTP, Telnet, BootP/DHCP as well as WEB server support.

2.2 Descriptions - Major Razor Components

2.2.1 PBOOT

2.2.1.1 Initial load (bootstrap) of the Razor platform

Let us begin by assuming that a prototype board that just arrived from the factory and testing is about to be started. At this time the board looks as follows:



Please note that initially there is nothing in Flash and that there are no software packages available to run the ADP (the Advanced Debugger Port that interfaces with host debug software) or the RS232 ports.

In order to get the first version of Pboot into the board, a connection is made between the host Unix¹ system and the board's JTAG hardware port (or the BDM port, if a Motorola processor is being used). This will allow host commands to be issued that will first download the Pboot version that is to be tested and then allow low-level debug software (using a program such as IBM's RiscWatch) to test the downloaded Pboot version.

The board's Flash memory file system should now be established and should contain a working Pboot version.

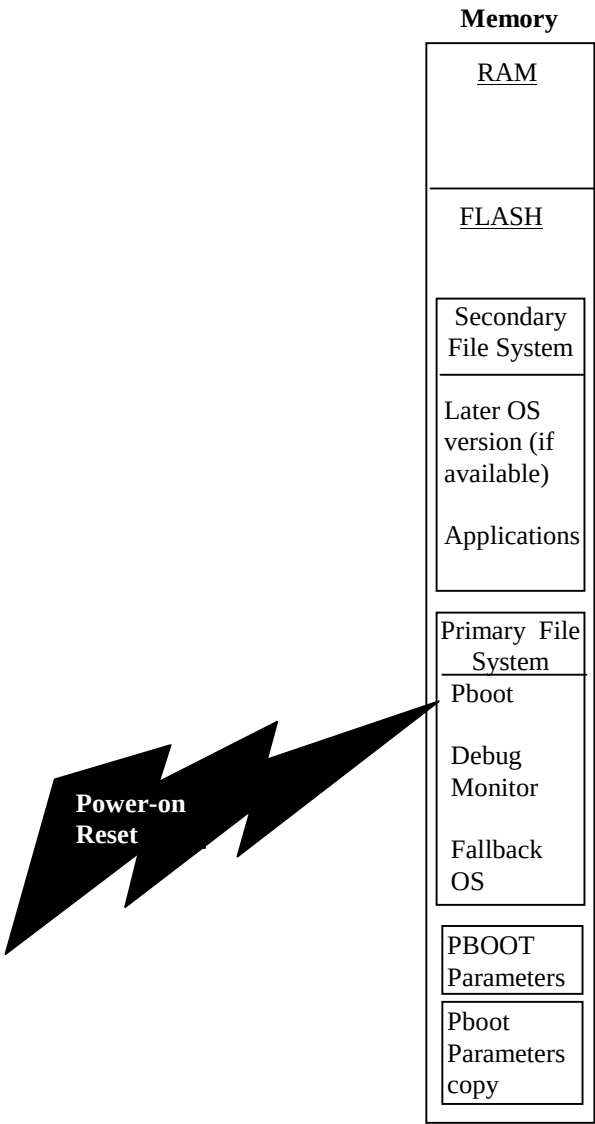
The next step is to execute the Pboot routine that will load a number of software packages that will allow the testing/debugging process to continue.

¹ Note that the majority of debug functions are also available under PC Windows. A Unix system is however required for program generation.

Before proceeding with the debugging/testing process, let's go back and take a closer look at the Pboot process.

Depending on the processor type used, the first instruction executed when power is first applied to the processor will either be a branch (jump) instruction that directs the processor to the Pboot routine or the first instruction of Pboot. The illustration below depicts the latter case.

Pboot resides in Flash memory and when initiated it will begin to execute out of the Flash memory in which it resides. The execution of Pboot will cause Pboot to eventually load itself into RAM memory.



Pboot will now access the Primary File System located in Flash memory and the Fallback OS and Debug Monitor² will be loaded into RAM memory (i.e., the entire Razor operating environment). In the event that the Secondary File

² Also referred to as the ROM Monitor

System contains a later version of the OS (this would be the case if the Management System's central processor has a more up-to-date version of the OS, see below), this version will be loaded into RAM memory. At this point the applications residing in the Secondary File system will be loaded into RAM and they will be initiated.

The contents of the Primary File System are initially burned into Flash memory at the factory. At this time, the contents of the Secondary File System are blank. The first time that a board is inserted into an operational embedded system environment and power is applied represents the first cold-start of the board in operation. The Fallback OS will be loaded from the Primary File System as described above after which the Management System's central processor will contact the board via the link and download a more up-to-date version of the OS if such a version is available. It will also download the current versions of all applications software that are applicable to the board in question. These versions will be written into the Secondary File System (if Flash memory capacity is available, otherwise a clean-up mechanism will start to make place for the new versions) and will be accessed when subsequent bootstraps are performed. During every subsequent bootstrap the Management System's central processor will download newer OS and/or applications software, if available - these newer versions will then be written into the Secondary File System³ whereby any older versions will be overwritten.

2.2.1.1.1 Razor's debug environment

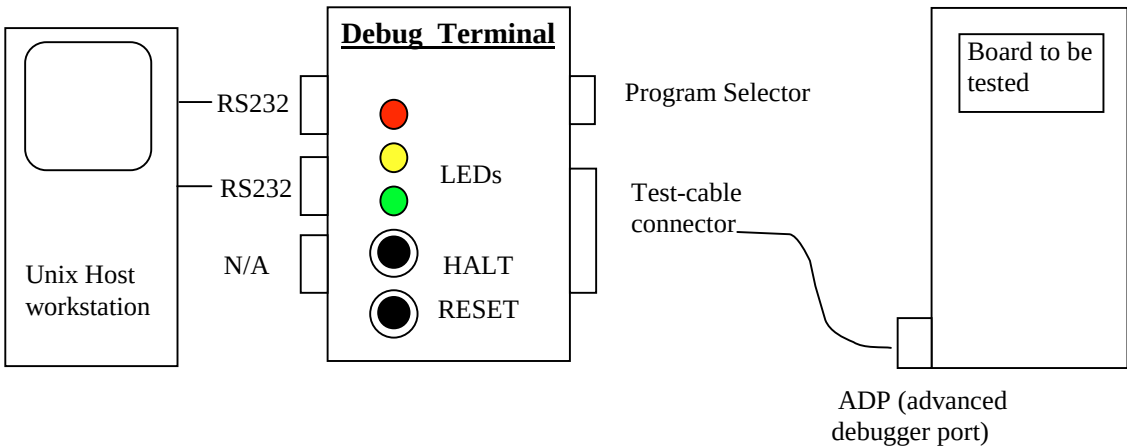
Now back to the debug/test phase. The description in the previous section deals with the bootstrapping of a board that is placed into operation, i.e., installed in an operational embedded system environment. Let us now look at the bootstrap variations that are available when testing/debugging is to be performed.

Assume that Pboot has been successfully downloaded to the board and seems to be operating OK.

Let us now continue with the boot process for the purpose of debugging/testing the OS, i.e., the entire Razor operating environment, the applications software and for want of a better term, perform basic low-level hardware-oriented testing, i.e., test certain applications-related hardware functions.

³ Note that both the Primary and Secondary File Systems may contain both program and data files, e.g., a typical data file might contain FPGA data.

A debug terminal is available for debugging purposes.



The debug terminal may be used to control the extent to which the bootstrap process will progress. At some point in its execution, Pboot will interrogate the position of the program selector (see diagram of the Debug Terminal above) and the depending on the program selector's position the process will proceed as follows:

Program selector position	Pboot action
0	Pboot will proceed to load everything in the Primary File System, i.e., the Razor OS, the Debug Monitor and anything else that happens to be residing in the Primary File System into RAM. Pboot will then activate the OS, which will inspect the contents of the Secondary File System. If there exists a different OS version in the Secondary File System, that version will be loaded into RAM and control will be passed to it. Device drivers that are stored in the Secondary File System will be loaded into RAM. Any applications found in the Secondary File System will now be loaded into RAM. The applications will now be activated. Note that although the Debug Monitor was loaded into RAM as part of the above process, it was not activated (instead, the OS was activated) and as a result, the host Single Step program cannot establish contact with the Debug Monitor at this time. Neither may any Pboot Commands be given to the Command Line Interface via a host terminal emulator – such as TIP, Kermit, or any other suitable VT100 emulator. Also note that the OS Monitor is active and may be contacted from the host via the program RPconsole. The Watchdog is activated by Pboot before the OS is loaded into RAM. May be disabled if required.

Note that this is equivalent to the situation in which no Debug Terminal is connected to the ADP, i.e., in the case where the board is mounted in the magazine of an operational embedded system environment.

- 1 Pboot will proceed to load everything in the Primary File System, i.e., the Razor OS, the Debug Monitor and anything else that happens to be residing in the Primary File System into RAM.

Pboot will initiate the Pboot Command Line Interface that will allow the contents of Pboot parameters to be inspected and changed by the host terminal emulator (e.g., TIP). The Command Line Interface also allows programs such as the bootstrap program, the Debug Monitor and any configured FPGAs (Field Programmable Gate Arrays, usually associated with a particular application) to be written over by new software versions.

Note that the above only applies to parameters and programs residing in the Primary File System.

Neither the OS nor the Debug Monitor will be activated. Applications will not be loaded into RAM.

The Watchdog is disabled by Pboot.

- 2 Pboot will proceed to load everything in the Primary File System, i.e., the Razor OS, the Debug Monitor and anything else that happens to be residing in the Primary File System into RAM.

Pboot behaves exactly as in the case of program selector position 0 with the following exceptions:

- the Razor OS is activated by the Debug Monitor and the Debug Monitor may therefore be contacted from the host program Single Step,
- the OS monitor may only be contacted via the RPconsole program in the host, and
- the Watchdog is disabled.

If the ADP adapter plug is not connected, as for example in situations where the board is installed in an actual operational embedded system environment, Pboot will react as in the case of program selector position 0 in the table above.

2.2.1.1.2 Summary of Razor's test/debug facilities

The following tables summarize a number of cases in which the test/debug facilities of Razor are applicable:

Case 1: Testing/debugging a new version of Pboot.

Case 2: Testing applications-related hardware functions.

Case 3: Programming FPGAs (Field Programmable Gate Arrays).

Case 4: Setting up the Primary File System.

Case 5: Testing the Razor OS.

Case 6: Testing/debugging application programs.

Case 7: Debugging timing, recovery, watchdog, etc. problems.

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
A new version of Pboot is to be tested/debugged for the first time.	This debug process is not sensitive to the program selector setting.	None, the board is completely empty of any software.	⊥ Nothing. Φ Programs such as Lauterbach or RiscWatch are used to introduce the first copy of Pboot into the board's Flash memory via the JTAG or BDM hardware ports. These programs also allow Pboot to be tested and debugged. A socketed Flash prom is an alternative to using the JTAG/BDM hardware.

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
Certain applications-related hardware functions are to be tested.	Program selector is to be set to position 1 before initiating the board.	Pboot and the Command Line Interface are active.	⊥ Pboot's Command Line Interface. Φ The host terminal emulator will communicate with the Command Line Interface. Entering commands via the host terminal emulator will cause test programs to be loaded from the host and stored in Flash. They will then be loaded into RAM memory for execution. These programs will test certain hardware functionality and will communicate with the host via the Pboot BIOS functions that are identical to those used by the Pboot Command Line Interface to communicate with the host terminal emulator. After the process, all programs will reside in Flash unless deleted via a host terminal emulator command. Note that programs run under Pboot are not relocatable and that they must be run in specifically defined RAM addresses. Note also that all programs located in the Primary File System that are assigned the file extension "PBA" will be automatically activated by Pboot. Such programs will remain active until deleted by host terminal emulator command.

Note that the above comment relating to "PBA" programs is independent of program selector position!

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
A number of FPGAs (Field Programmable Gate Arrays) are to be programmed.	Program selector is to be set to position 1 before initiating the board.	Pboot and the Command Line Interface are active.	⊥ Pboot's Command Line Interface. Φ Entering commands via the host terminal emulator will cause the FPGA programs to be loaded from the host and stored in Flash. They will then be loaded into RAM memory and executed whereby the FPGAs will be initiated. If desirable, a program can then be loaded that will test the FPGAs that were just initiated.⁴ After the process, all programs will reside in Flash unless deleted via a host terminal emulator command. Note that programs run under Pboot are not relocatable and that they must be run in specifically defined RAM addresses.
The Primary File System is to be set up.	Program selector is to be set to position 1 before initiating the board.	Pboot and the Command Line Interface are active.	⊥ Pboot's Command Line Interface. Φ Entering commands via the host terminal emulator will cause the Debug Monitor and the OS (the entire Razor operating environment) to be loaded and saved into Flash. They will not be initiated at this time unless a host terminal emulator command is given to run them.

⁴ The FPGA Loader and the FPGA image may be stored as separate files, which means that FPGA data can be updated without having to run the make system to regenerate the Loader program.

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
The Razor OS is to be tested.	Program selector is to be set to position 2 before initiating the board.	Pboot will be loaded into RAM memory whereupon everything in the Primary file System will be loaded into RAM memory. This means that RAM will contain: Pboot, The Command Line Interface, The Debug Monitor, and the OS including the OS Monitor. The activation of the OS will cause any *.OSE test programs to be initiated from Flash. The Debug Monitor must be used in the event that an error occurs during the running of these test programs. Upon loading all of the above into RAM, Pboot will give control to the Debug Monitor which will in turn activate the OS.	⊥ Debug Monitor and the OS Monitor. ΦThe host terminal emulator can communicate with the OS Monitor. Access to the Debug Monitor is provided by the source-code debugger, SingleStep.

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
Application programs are to be tested.	Program selector is to be set to position 2 before initiating the board.	Pboot will be loaded into RAM memory whereupon everything in the Primary file System will be loaded into RAM memory. This means that RAM will contain: Pboot, The Command Line Interface, The Debug Monitor, and the OS including the OS Monitor. Upon loading all of the above into RAM, Pboot will give control to the Debug Monitor which will in turn activate the OS. At this point the Host may issue an RPlload command to load application programs from the host to the Flash Secondary File System. This action will cause the programs to be automatically loaded into RAM and initiated. Testing can now start.	⊥ Debug Monitor and the OS Monitor. Φ The host terminal emulator can communicate with the OS Monitor. Access to the Debug Monitor is provided by the source-code debugger, SingleStep.

The function to be tested or the operation to be performed	Debug Terminal program selector position	Resident, active programs in RAM at this time	The debug programs run in the Razor environment [⊥] and in the host ^Φ
Debugging of problems such as those relating to timing, recovery, watchdog, etc. Generally applies to problems of a thus far indeterminate nature, i.e., the cause and/or the failing routine and/or hardware have not as yet been isolated. Problem requires dump analysis.	Program selector is to be set to position 0 before initiating the board.	Pboot will be loaded into RAM memory whereupon everything in the Primary file System will be loaded into RAM memory. This means that RAM will contain: Pboot, The Command Line Interface, The Debug Monitor, and the OS including the OS Monitor. Upon loading all of the above into RAM, Pboot will give control directly to the OS. At this point the Host may issue an RPload command to load application programs from the host to the Flash Secondary File System. This action will cause the programs to be automatically loaded into RAM and initiated. Alternatively the Secondary File System already had application programs stored in it, in which case they will be loaded by the OS and initialized. Testing can now start.	⊥ The OS Monitor. ΦThe host terminal emulator can communicate with the OS Monitor.

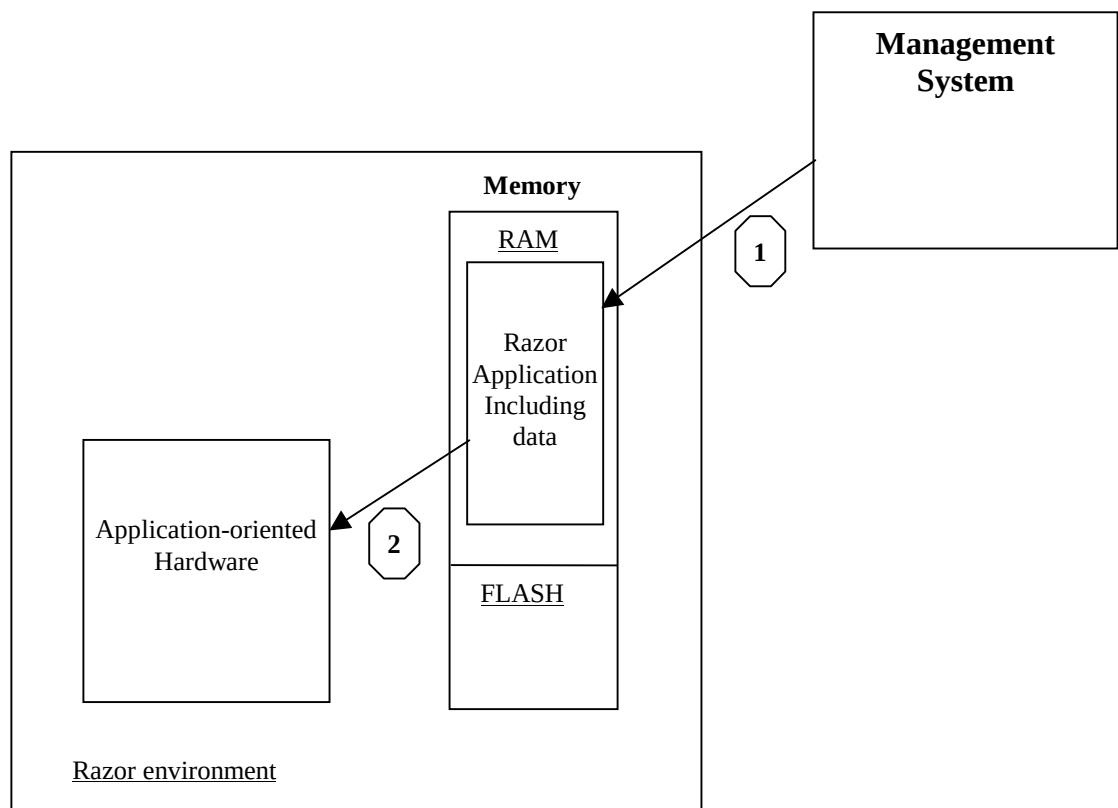
2.2.2 Generating A Razor Program Unit

2.2.2.1 Introduction

To begin with a Razor application looks like any other computer application in that it includes both instructions and data. A Razor application or “program unit” is always downloaded from a management system to the particular Razor environment in question. Generally the Razor environment is contained within the confines of one electronic board but may, under certain circumstances, span a number of boards (see section 1.6.1).

For the most part a Razor application will involve the loading of application data into some sort of application hardware located on the board. This hardware may for example consist of one or more FPGAs (Field Programmable Gate Array) and/or DSPs (Digital Signal Processor).

The process is depicted below.



First (1), the application - including its instructions and data - is loaded into the Razor environment’s memory and then (2); the application will proceed to download its data into the application-oriented hardware of the Razor environment.

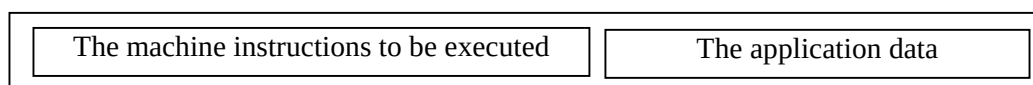
At this point, the hardware will be initiated and will run so to say, free-wheeling - that is, without the benefit of the application’s intervention. The hardware will continue to run and will only report to (interrupt) the application in the event of the occurrence of a fault or when the hardware requires special support.

2.2.2.2 Program unit generation methodology

All Razor program units have the file extension "chk", i.e., these files have the form *.chk.

These files may be further encapsulated into Kongsberg files having the file extension "rpc" for the purpose of storing them in the AXE CP.

A self-contained program unit basically includes the instructions that are to be executed plus the data that is to be used in connection with instruction execution. There is a whole spectrum of applications which are not self-contained and therefore rely on the acquisition of external data – be it from one or more databases, from an operator, or from data that is collected by some sort of process that is being controlled. In the case of Razor applications one is only concerned with applications that are completely self-contained. A Razor application therefore looks like:



The generation of a program unit may take various forms, and is often dependent on a particular company's organization of its development and implementation processes and/or on the complexity and variation of the applications involved.

For example the group that develops and programs an application's instructions could feasibly introduce the "live" data required for the execution of the application in the field (e.g., at a customer site). This would however entail that the development staff must be aware of the particular market and its customers' requirements. This may work for a small organization but not for a company having a global presence.

Somehow the development organization must supply its various market-oriented units with applications software to which, particular market-oriented data, can be linked.

This can be accomplished through the development organization's release of application source-code to the various marketing/sales organizations and allowing them to add the required data and then to compile the programs so that self-contained program units can be produced for the particular markets/customers involved. The primary drawback with this approach is that control is completely relinquished to these organizations and centralized technical support can hardly be maintained thereafter.

The method that was selected and which is incidentally used throughout Ericsson for the generation, and field-introduction, of software units is the following.

Basically three organizational entities are defined for every product:

- The application development organization
- The marketing/sales oriented organization
- The configuration control organization

An Ericsson-wide file format has been designed for the generation and maintenance of program units – the RPDOUT format, which has recently been released as an open standard.

Header	Instruction Code	Data	Dynamic File system	Environment Variables	Checksum
1	2	3	4	5	6

The development organization supplies compiled relocatable code and data for sections 2 and 3. The marketing organization supplies the data and variables required for sections 4 and 5. The configuration organization puts it all together and creates the Header and checksum fields. The program unit may now be released to a particular customer site.

In particular please note sections 4 and 5, i.e., the sections “Dynamic File System” and “Environment Variables”.

The dynamic file system allows as much or as little data to be added to the finished program unit as is required by the particular customer application.

The environment variables field is basically filled with information that “describes” the particular environment in which the application is to operate. The Razor operating system includes functionality that facilitates the interpretation of these variables and allows the application program to behave differently depending on that interpretation.

The above construction is extremely helpful in facilitating the field deployment of one and the same code in order to fulfill a variety of customer requirements.

As a result, program units⁵ can be delivered that:

- are tailor-made to fit the particular requirements of a particular market and/or customer, and
- can be supported by a centralized organization as if only a single code-variant existed.

⁵ A program unit is a self-contained, executable program.

2.2.3 Memory Management

2.2.3.1 Why memory management - the key issues

The main functions handled by the Memory Manager are:

- The administration of free memory
- The administration of cache memory
- Memory protection
- Address translation and virtual memory mapping

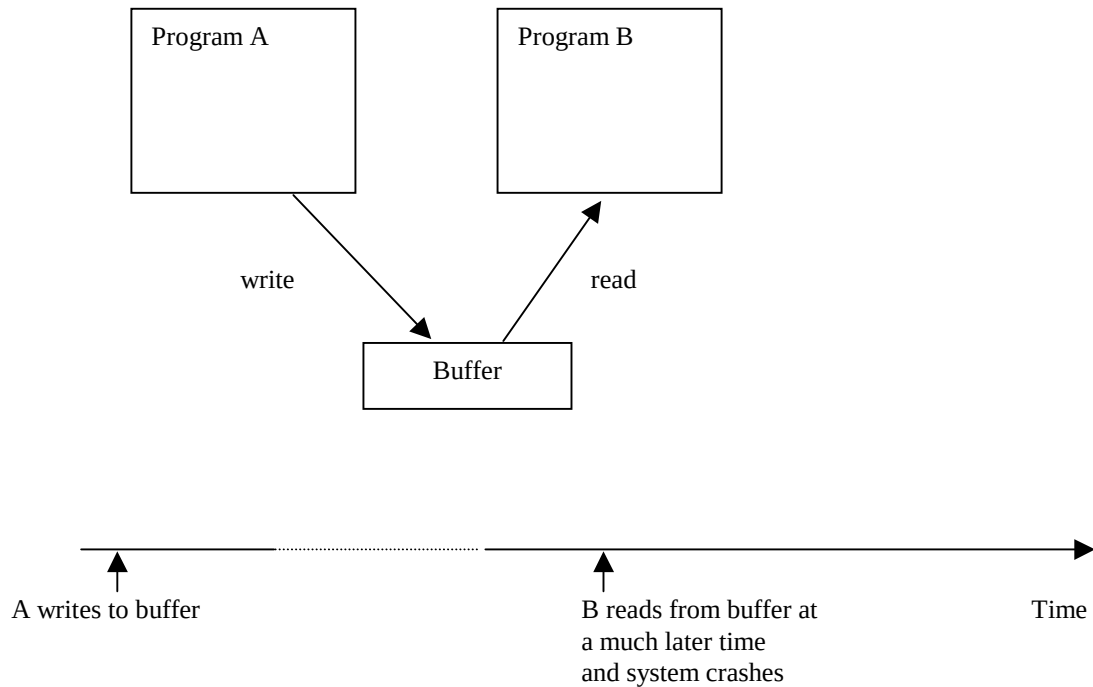
The operating system's memory management service provides for the dynamic allocation and freeing of memory. Access to memory-mapped hardware is also made possible by the memory management service. Memory management provides the ability for data to survive the restart of an application and/or the operating system (Note: This is not possible in the event of a power-up cold start).

The principal users of memory management services are the program loader, the PCI sub-system and applications that make use of persistent data functionality and/or applications that use great amounts of memory. Ordinarily, normal applications do not have any reason to make use of the services provided by the memory manager.

1. Fault detection – a fault-tolerant system is based on the principle of providing maximum fault detection - in this case it's memory protection violation detection.
2. Reduction of the probability of fault propagation and the provision of near one-hundred-percent fault localization, i.e., the ability to trap a fault before it propagates to other program locations (thereby giving incorrect error details). Expressed in somewhat different terms, this boils down to the ability to point to the failing instruction with more or less one hundred percent certainty.
3. The improvement of software locality, i.e., the ability to point directly to the line of source-code at which an error has occurred.
4. The reduction of memory fragmentation, i.e., the avoidance of unnecessary "holes" in memory.
5. The ability to support symbolic debugging aids.
6. The provision of efficient interrupt handling, i.e., the avoidance of executing too many instructions when processing interrupts.

Fault propagation is very likely the reason for the expenditure of many unnecessary programmer-hours in connection with fault localization and correction.

A simple example of fault propagation.



The dump that is generally provided at system crash time points to the fact that Program B is the culprit when in reality Program A wrote an incorrect message to the buffer which was not detected - the fault appears to have propagated in both time and locality.

The implementation strategy

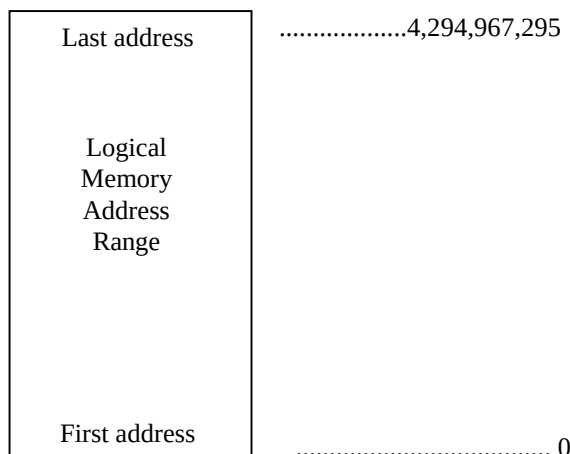
Memory management is implemented such that it provides:

Memory management functionality	Addresses issue number
Memory protection and restart dump addressing. Memory management is implemented so that any and all memory protection violations are detected. Furthermore, all dumps start from the same start address which makes debugging simpler and allows for a correspondence to be established between the program unit's addressing scheme and that of the program's source code.	1, 2 and 3
Memory is virtually mapped in 4K or 16K byte-blocks. To minimize the occurrence of memory "holes" or fragmentation. In the worst case, it is possible that only a portion of a block is used which could be considered as the cause of a "partial hole". A large virtual address space simplifies address planning to a point where programs can nearly always be loaded at a preselected address.	4 and 5
One single address space for all programs. Does away with the need for ordinary time-consuming context switching mechanisms employed by conventional operating systems that support virtual memory addressing.	6

2.2.3.2 The Razor Memory Manager

Memory addressing

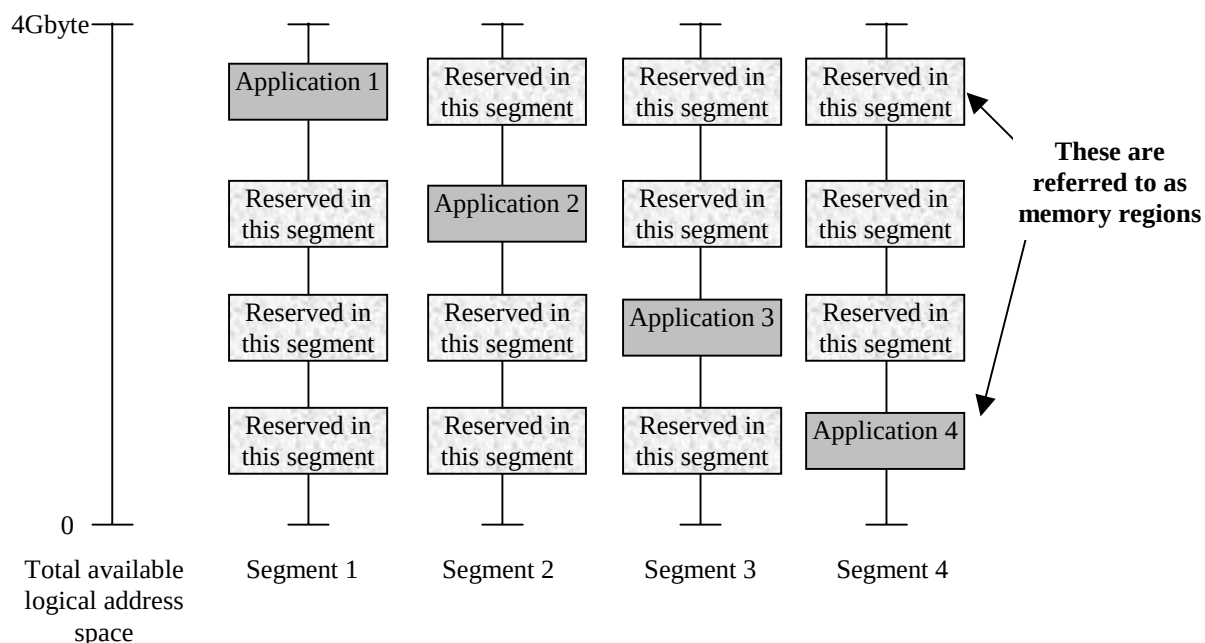
Memory addresses consist of 32 bits which means a logical address space of $2^{32}-1$ or an address range from address 0_{10} to $4,294,967,295_{10}$.



Actually, Razor environments are configured with much less physical memory than that shown above. Typical Razor application environments have been developed for physical memories that range from 16 to 128 Mbytes.

A Razor environment that is configured with a RAM memory of 128Mb, effectively utilizes $134,217,728/4,294,967,296$ (i.e., $128\text{Mb}/2^{32}$) or only 3.1% of the total addressable memory space available.

The illustration below will help to clarify the situation.



See the next page for a discussion of memory segments and regions.

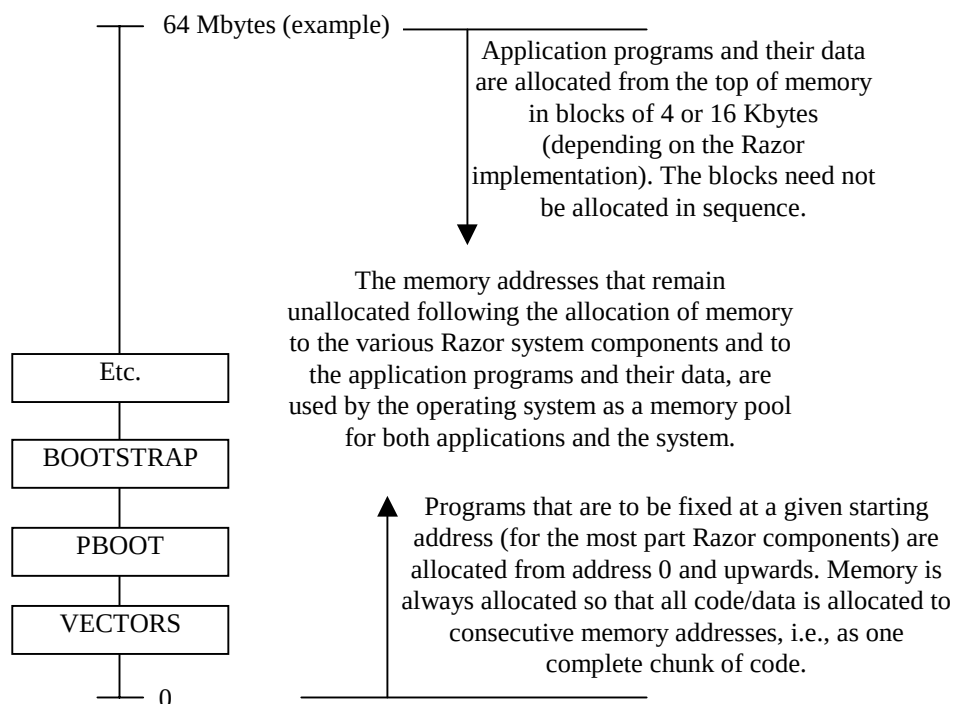
To begin with all applications and their data share the same logical address space of approximately 4 Gbytes. The total available physical memory is much less than 4 Gbytes, i.e., somewhere between 16 and 128 Mbytes.

Each application program is allocated its own memory segment (or its own logical memory address space) that is 4 Gbytes in size. Memory clashes between the different application programs and their data is avoided as follows. Each application is allocated one or more memory regions for instructions and data within its own segment. The regions that are allocated to a given program in its segment are marked as “reserved” in the segments allocated to other applications.

In this way, applications and their data can effectively share one and the same logical memory space and avoid any interference with one another.

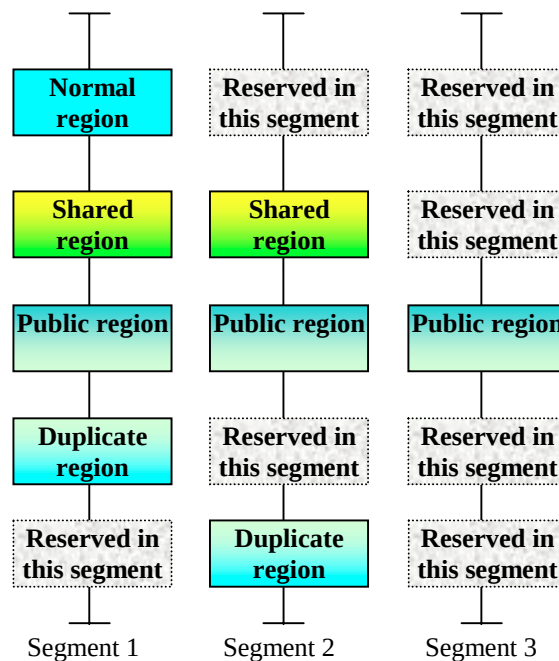
Physically the above allocation process is conducted as follows:

The diagram below is an illustration of the allocation of physical memory to applications, data and Razor system components.



Memory is managed through the use of two free-page lists. One list operates from the top and the other from the bottom of the free physical memory available. The first list is used for the allocation of memory to logically addressed memory regions and the second list is used for the allocation of consecutive physical memory regions. In both cases, memory is managed in increments of pages - the allocation strategy described above serves to reduce free memory fragmentation.

Various types of regions



Four types of regions exist:

- The normal region – the address range within the 4 Gbyte address space is reserved in all other segments, i.e., reserved so that the address space is not used by other programs.
- The shared region – which allows one or more segments to share a common address space. Note that those segments that have not declared the region as shared will reserve this address space to avoid committing a memory access violation. The common memory area is seen by all applications having access to the area as having the same memory attributes. The memory allocated for shared regions is not freed until the last region referring to the memory is deleted.
- The public region – is defined in all segments. This means that all processes can access the region. As an example, a common signal pool can be created as a public region. It is generally recommended that regions be duplicated or shared if they are to exist in many segments since public regions only remain in the system until the next warm restart, unless already deleted in all segments. Persistent regions remain even after a warm restart.
- The duplicate region – may be duplicated in one or more segments. Note that those segments that have not declared the region as duplicated will reserve this address space. Note also that duplicate regions make efficient use of the fact that their respective attributes need not be the same. For example, if segment 2 contains a duplicate of a region in segment 1, then cache may be turned off in segment 2 and turned on in segment 1. This would allow a checksum control of the region to be performed (if performed in region 2) without affecting the contents of cache and therefore have no impact on execution performance in region 1. The duplicate region can be situated in the same or in a different segment. The memory allocated for

duplicate regions is not freed until the last region referring to the physical memory is deleted.

Handling cache memory

Generally a computer central processor will have access to some sort of very fast internal memory that is generally referred to as a cache. The cache is used for the storage of recently used data and the corresponding “low-speed” memory medium (such as RAM) in which the data and/or instructions actually reside will be updated as needed. Caches can be configured to operate in different modes:

- Turned off

Cache is not used at all.

- Write-through

The “slow-speed” memory (e.g., RAM) is updated at the same time as the data in cache is modified. This implies that memory must be accessed via the CPU bus for every cache write.

- Copy-back

The “slow-speed” memory is only updated when the contents of a particular cache address (or addresses) must be vacated (purged from cache) to make room for other data. This makes for the fastest form of execution since the CPU bus is not used until the RAM absolutely must be updated.

The above modes can be set separately for different memory regions.

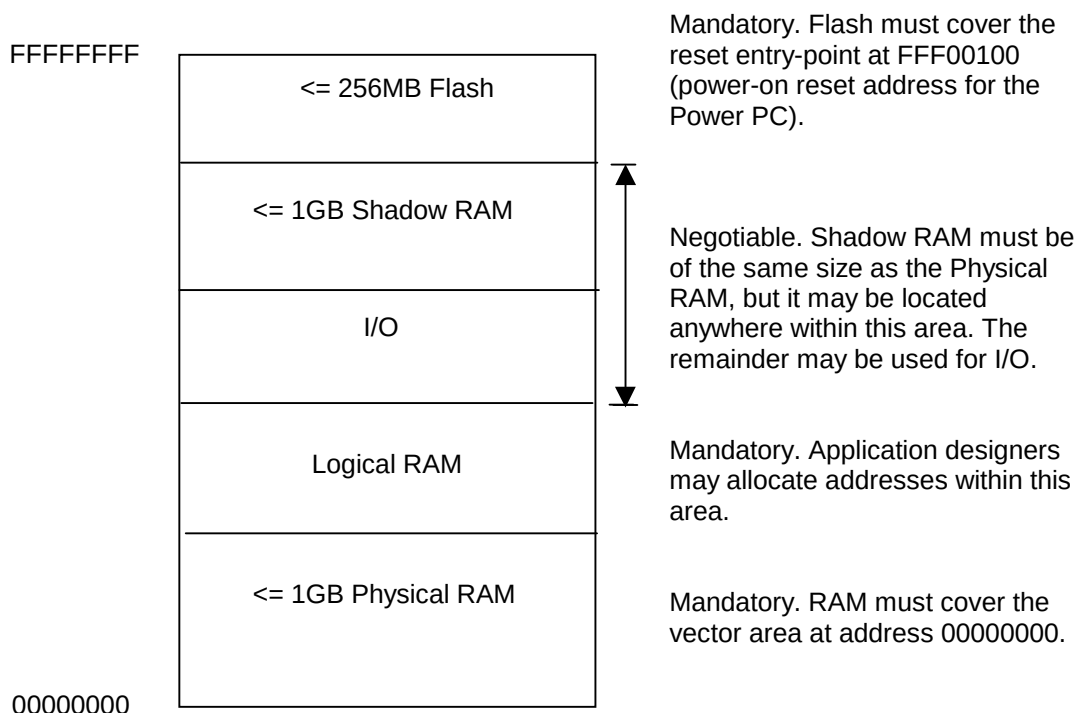
Address translation and virtual memory mapping

A memory map may be considered as a plan for the management of memory resources. The map indicates how different areas (addresses) may be used for different purposes. Discussions of the memory map in connection with the memory manager’s address translation mechanism almost always necessitate that memory be viewed as a virtual memory space.

Virtual memory space contains one or perhaps several areas reserved for applications, memory-mapped hardware, and I/O buffer-space. The disposition of these areas as well as the actual memory map varies from project to project.

Discussions of memory mapping often include the term “physically mapped”. This refers to an area in the virtual memory space in which logical addresses are exactly the same as the physical addresses. These addresses are used for example in DMA transfers that involve external devices that do not use address translation.

The illustration below depicts a typical memory map.



Logical memory

Physical memory is often a scarce resource and platforms are seldom equipped with as much physical memory as that contained within the addressing limits of the particular computer processor used. Using logical memory is pretending to have as much memory as is addressable, and then so to speak, letting the memory manager ensure that the right physical memory addresses are mapped to the used logical addresses.

A logical address is one that refers to another address in RAM. To access a logical address one needs a memory manager and a CPU that supports logical-to-physical address translation.

Generally speaking, one cannot have more data in a machine, than that which actually fits in the machine's physical memory. However if some sort of swapping⁶ technique is used then applications and their data can be much, much, larger than the actually memory-span available. This is the basis of "virtual" memory machines in which the apparent size of memory is many times that of the actual physical memory available.

Large virtual memory schemes allow applications the possibility to select address spaces that do not collide with other applications. The relocation of programs to new (logical) addresses is almost effortless, and it makes it

⁶ A method whereby certain memory addresses are "swapping out" to an external unit such as a hard disk so that other programs and data can be "swapped into" memory for execution.

possible to run programs in memory spaces that are non-contiguous (physical pages can be allocated from anywhere in the physical memory).

Virtual memory also decreases the risk of memory fragmentation. Memory fragmentation occurs when a region cannot be created due to the fact that there is no contiguous memory area large enough to allow the region to be loaded. This problem is resolved through the use of logical addressing - physical region addressing requires that an address be mapped to the same physical address.

Note that Razor environments only have one virtual address space in which one logical address only refers to one physical address, unlike some other memory managers (such as that in UNIX).

I/O-space

A special type of region for which the “iospace” attribute is set. Note, this is a region for which no RAM is allocated. It is occasionally necessary to access certain memory or hardware registers that lie outside of RAM.

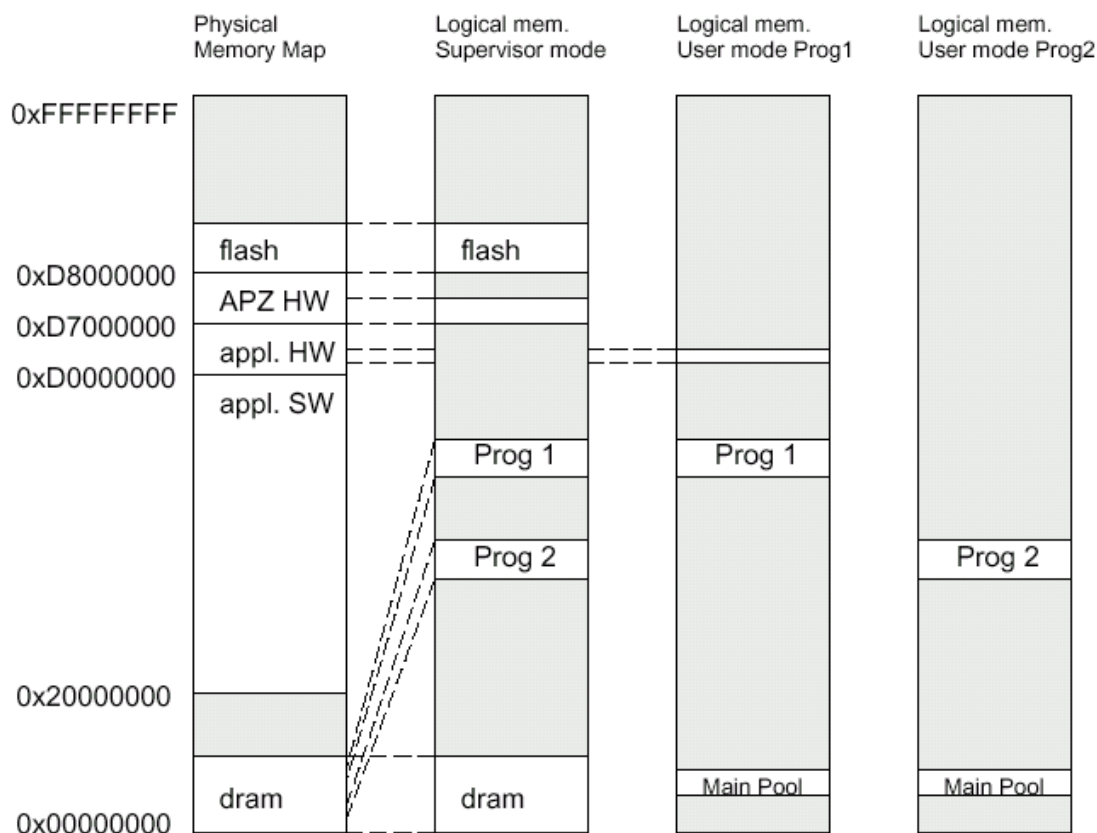
Once virtual mode has been turned on in the CPU, the only accessible memory addresses are those for which the memory manager has constructed page tables. In addition, all logical addresses have thus far only been mapped to physical addresses situated in RAM.

Persistent data

A region can be defined as being persistent. This means that the region will survive a warm restart of the operating system. At restart the consistency of the memory manager internal data structures are checked using checksums.

There is no consistency check of the contents of an application’s persistent memory region. If needed, it is the responsibility of the application.

An example of a memory model



The physical memory map shown in the above example includes a number of dedicated memory areas for DRAM, application programs, H/W and Flash. All of these entities except for the application program area have their representation in physical devices addressed by chip select logic in the address range shown in the figure.

Since hardware-supported address translation is utilized by the MMU, a logical memory map can include memory at addresses where no physical devices exist. Typically, this means that an application's program and data store reside in the area for application software and each page in the logical memory space of that application is translated into a physical page in the physical DRAM. Note that these physical memory pages need not be consecutive, even if the logical memory region is.

It is the application's responsibility to map its own hardware into its address space using the Memory Manager user interface – this must be done before the hardware is accessed. In addition, all programs in the system are accessible from supervisor mode. The operating system, as well as any autostarted programs, executes in supervisor mode. Routines installed as interrupt processes, OSE BIOS devices or REOS device handlers.

A typical application memory map only shows the actual program, the H/W areas that may have been mapped by the application and the main pool.

An application's program and data stores are write-protected against access from other applications. The main pool may be shared among a number of applications, in which case there is no memory protection between applications via signal buffers. This construction may be chosen for capacity reasons. When signals are sent from an application to the operating system/autostarted program (or vice-versa), the signals are copied from one pool to the other by the operating system.

2.2.4 Restart/Recovery

2.2.4.1 Background

A number of factors are fundamental in that they have been crucial in the design of the Razor platform fault-detection/recovery paradigm.

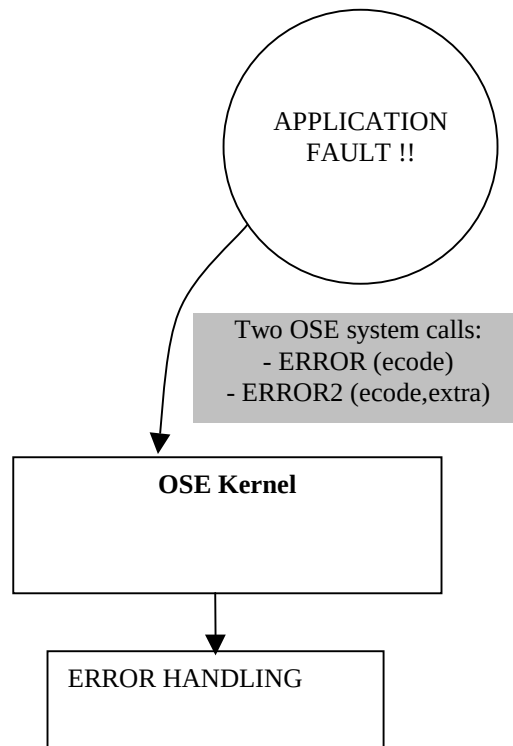
1. The Razor environment is to be fully operational within 5 seconds after the detection of an error, i.e., error detection and recovery may not take more than 5 seconds.
2. Error recovery is to follow a strict escalation process that is geared to quickly eliminating the possibility of attempting to restart an environment that includes hardware errors.
3. Within the time constraints mentioned above, to dump as much information as possible so that the faulty software and/or hardware entities can be identified with a high degree of certainty.
4. Three restarts within a span of 10 minutes will cause the management system to declare the unit as permanently “locked”, i.e., out of order.

Therefore, the occurrence of the slightest error will always lead to a restart dump (i.e., at least a primary dump) and the warm restart of the environment. If the warm start is successful then the error will be seen as being a temporary “hiccup” and processing will continue and the dump will be analyzed.

A fault can be so corrupting that the recovery mechanism itself is unable to function correctly, in which case additional dumps (secondary dumps) will be taken. The card may attempt to warm-restart a number of times after which a cold start will be attempted.

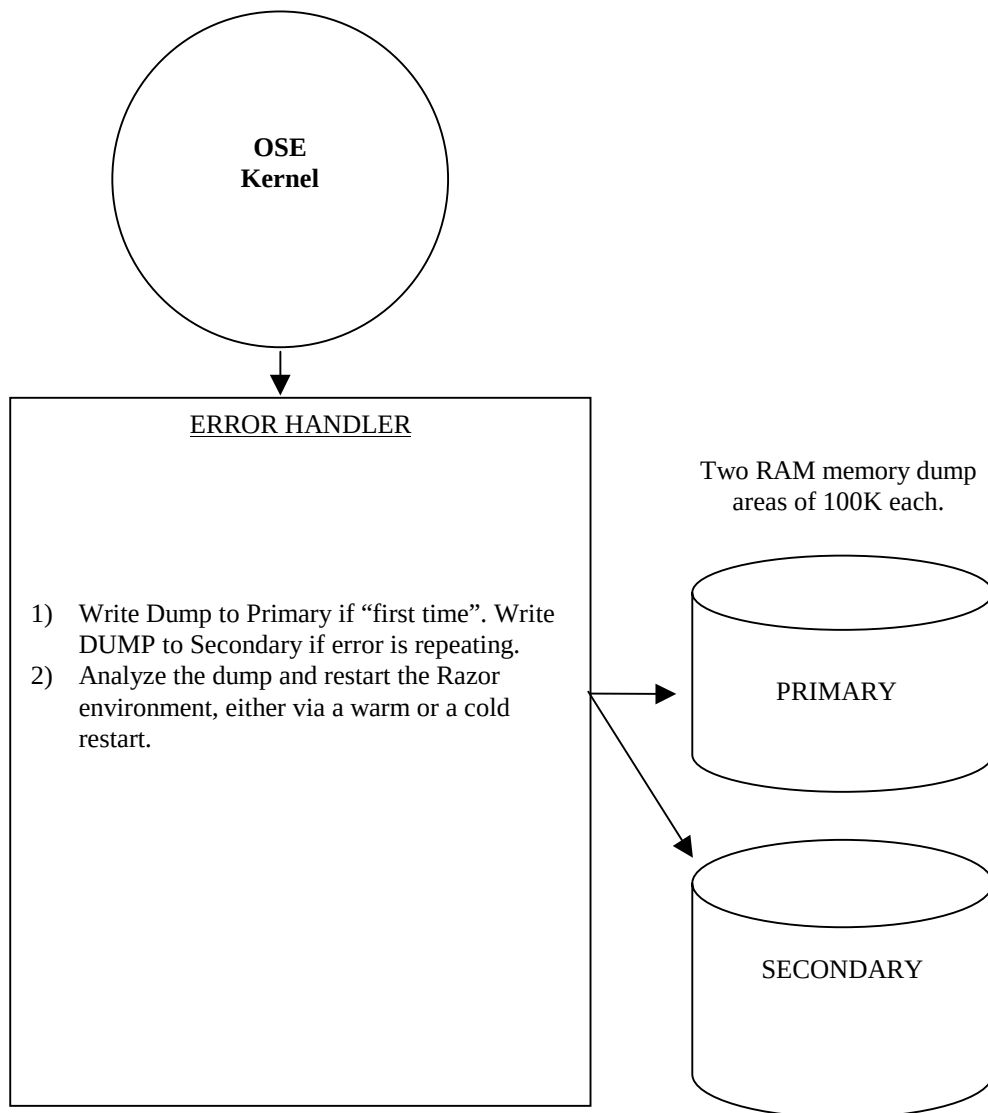
2.2.4.2 Introduction

Any error occurring during the operation of a Razor application will cause the immediate cessation of operation - control will be given to the Error Handler:



The Error Handler will attempt to determine whether the error may be recovered from and whether a warm or cold start is required. The guiding philosophy is that no Razor environment be allowed to perform its normal operational duties unless complete reliability can be guaranteed. The slightest error occurring in the environment must be detected and localized (i.e., identified and linked to the faulty code and/or hardware unit).

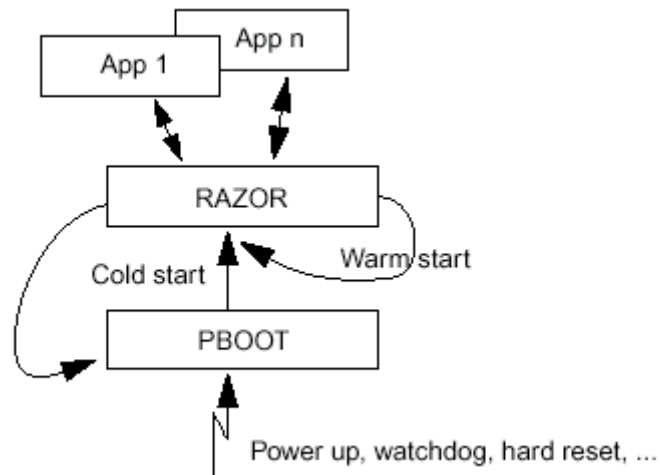
2.2.4.3 The Error Handler



The restart dump areas contain textual information as to the error that caused the system to stop and the status of the application/system at the time of the error.

The primary dump area contains information relating to the state of the application and the system at the time of error detection. Any spin-off errors (generally system errors) that may occur during Error Handler analysis or during the subsequent restart process will be dumped to the secondary dump area (secondary dumps contain less information per dump than does the primary dump).

Depending on the system's behavior during the dump analysis phase and whether or not a succession of dumps is produced due to a complex fault situation, the Error Handler will either initiate a warm or a cold restart of the environment.



Both warm and cold starts will cause the management system to experience a temporary link disconnect. The management system will generally react by sending an unlock command in the hope that the fault is of a transient nature.

Restart dump information is retained in RAM after the performance of a successful warm start. In the event of a cold start, RAM contents are essentially lost and only a small amount of information remains. This information may be fetched by the management system.

2.2.5 Program Handling

RAZOR contains functions for loading, start, stop and supervision of separately linked application programs. Programs can be loaded from an embedded file system or from a network server. Programs can be stored in the embedded file system or stored in RAM.

The embedded file system consists of the primary and the secondary file system. Programs in these file systems are handled differently.

Primary file system programs

RAZOR contains an auto-loader for OSE programs located in the primary file system as files with either the *.ose or *.auto extensions. Programs with the auto extension are only started by a fallback OS.

During OSE restarts, PBOOT is queried for programs that are to be locked in memory and started as OSE programs.

The auto-loader supports programs in the RPDOUT format (see section 2.2.2) as well as free format programs without special-purpose headers. For RPDOUT programs, program parameters found in the program image are set as block level environment variables.

Program properties for those started by the auto loader:

- Executes in supervisor CPU mode.
- Attached to and uses the system pool (same as RAZOR).
- OSE environment variables are set as described above.
- Created in a OSE block with the same name as the Surname parameter in program.def.
- No memory manager segment or regions are created.
- Linked to physical RAM addresses and therefore not relocatable.

Secondary file system programs

Programs located in the secondary file system can be of two types - autostarted and normal. Autostarted programs are started automatically by the RAZOR program handler without any intervention from management SW. This program type is suited to handling external communications (i.e., with the rest of the world).

Normal programs are not automatically started and must be started by management software using a suitable communication channel between the management system and the target.

Program properties for those started by the program handler:

- Executes in user or supervisor CPU mode as specified during program generation.
- Autostarted programs are attached to and uses the system pool (same as RAZOR)

- Normal programs are attached to and use the main pool (shared between normal programs)
- OSE environment variables are set as configured in the RPDOUT format.
- Created in an OSE block with the same name as the SUNAME parameter in program.def.
- Memory management segment and separate code plus data regions created.
- The code region is write protected.
- Linked to logical RAM addresses (not to physical addresses) and therefore relocatable.

2.2.5.1 The current program load strategy

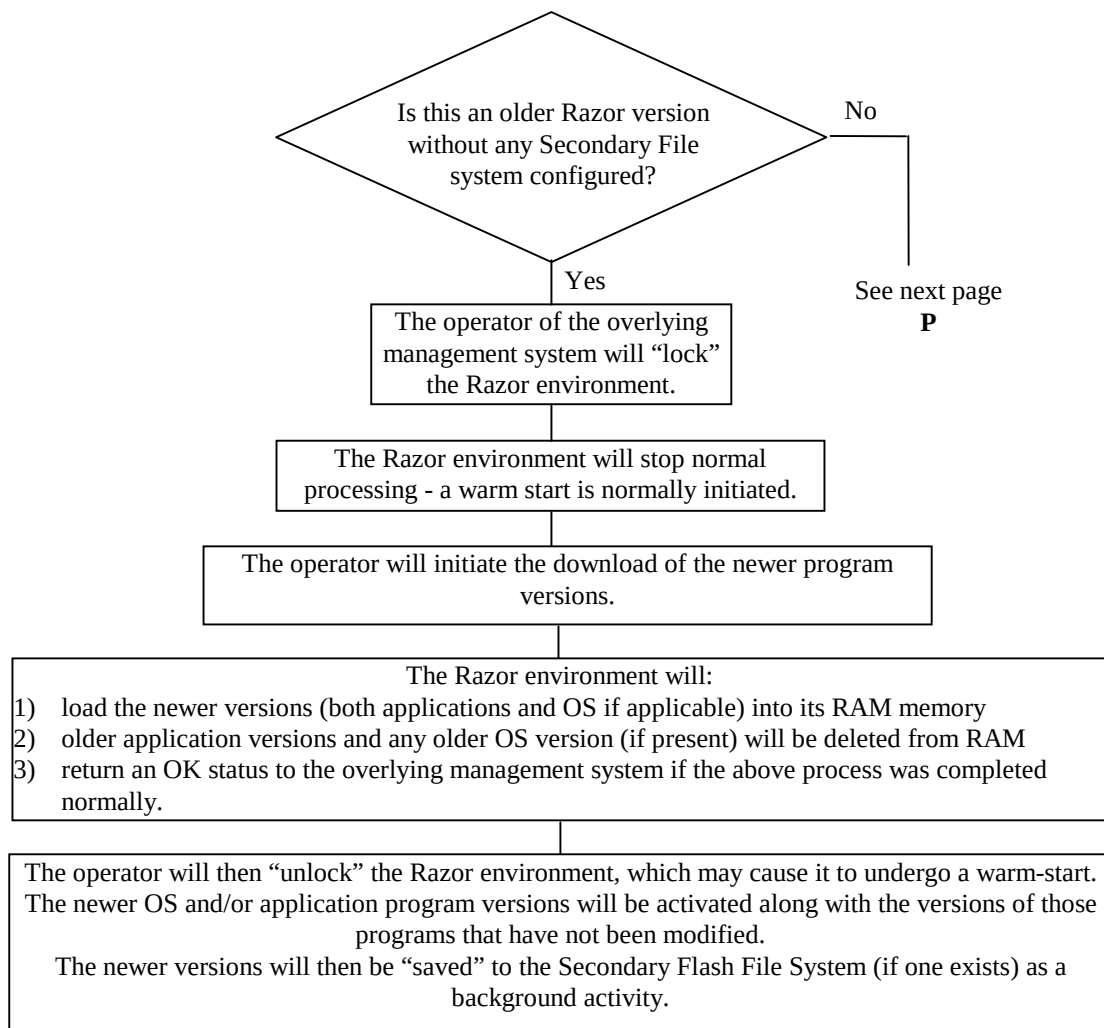
Program load is basically the process whereby new Razor versions and application programs are downloaded from the management system to the Razor operational environment.

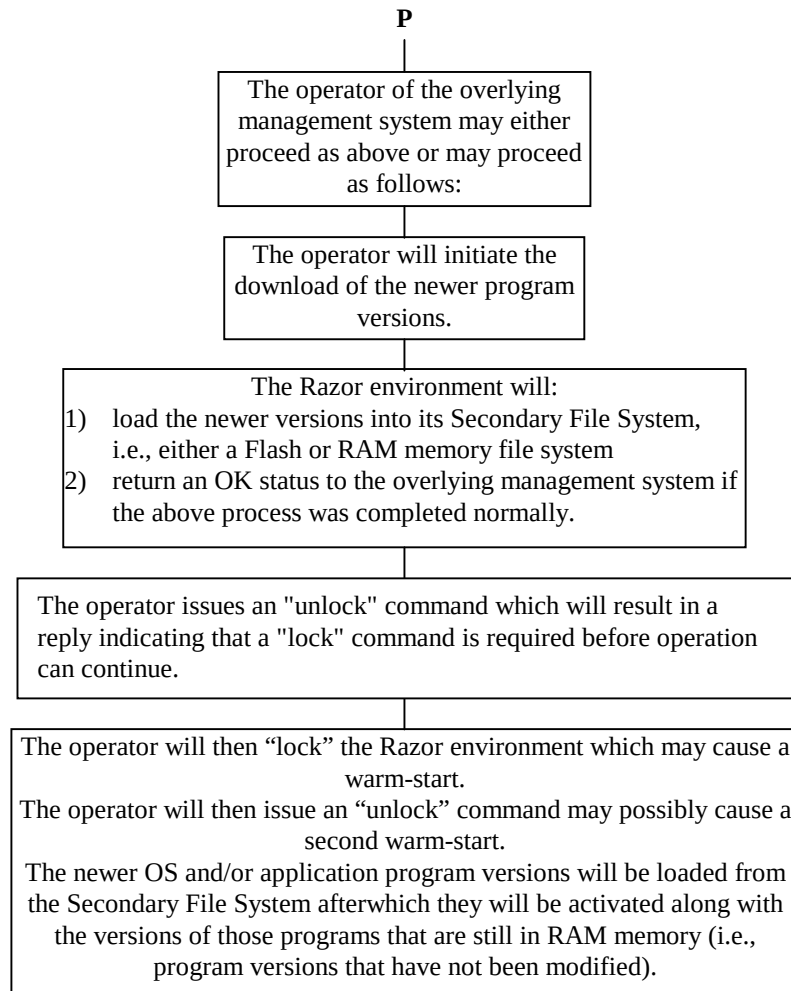
The current strategy for downloading new OS versions and application programs into the Razor environment is as described below.

The download process is a function of the version of the Razor environment in question. That is, whether it's an older version without a Secondary File System (either a RAM or Flash Secondary File System), or a more current version that includes a Secondary File System.

The Razor environment can assume one of two states. Either applications are up and running (the so-called "unlocked" state), or they're halted, i.e., not running (the so-called "locked" state).

The aforementioned states are the result of an operator command to either start or to stop the operation of the Razor environment.





2.2.5.2 A possible future program load strategy

Today's management systems are perceived as evolving towards becoming more open, being somewhat less proprietary and generally towards providing a more common and perhaps even "universal" interface.

Today's operators are faced with the management of an assortment of equipment that comes from a variety of telecom vendors and that provides a wide selection of services. Market factors require that future management systems allow the interconnection and interaction of a broad array of telecommunications equipment. Simply stated, a uniform or common management interface is what's needed.

Telecom vendors who wish to remain in the business will be forced to conform to these emerging market requirements. Requirements such as interoperability and interconnectivity which address rising customer needs in the area of cost-effective network operation and management.

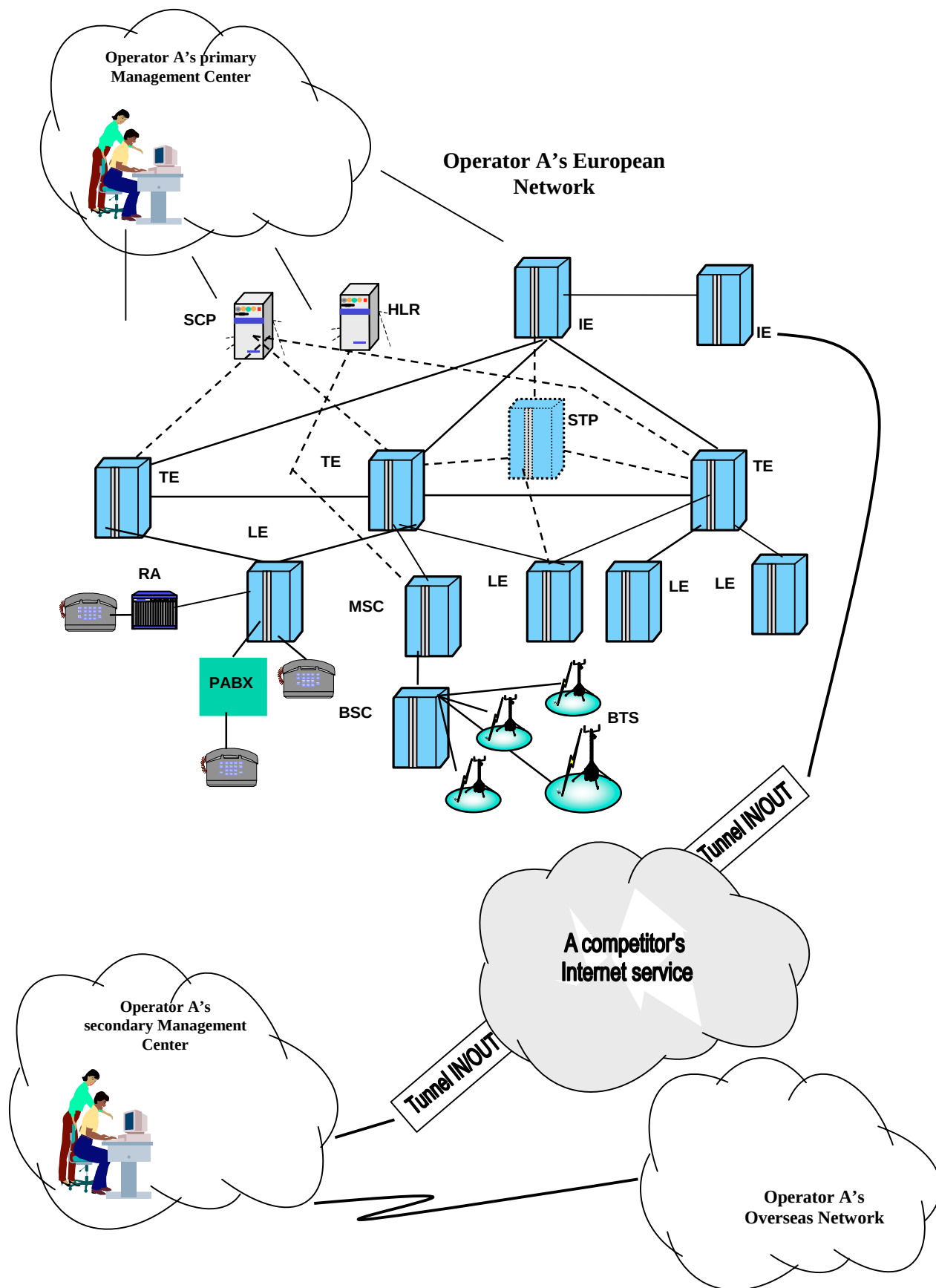
For the purposes of the discussion below, imagine a network that has the following management attributes:

- 1) Network management and operation is based on the use of standards such as Internet, Java applets and CORBA⁷.
- 2) Network management and operation is based on the premise that all program updates, billing information, statistical analysis, operations and maintenance activities, etc., can be performed from one or more centralized sites that are networked both to one another and to the customer-traffic network.
- 3) Management sites will be surrounded by firewalls to guarantee security. The use of external Internet services, i.e., Internet links that are the property of other operators, are used only in conjunction with tunneling. Tunneling is a method of encapsulating data sent over an Internet connection so as to increase its data integrity and therefore its confidentiality.
- 4) Site operations personnel need no longer type in specialized cryptic commands to control the operation of the various network elements. They will simply log-on to a suitable Internet server which will download a Java Applet. The applet, serving as the operator's graphical user interface to the command system, will execute locally in the operator's workstation. The operator will then click on the available fields and fill in the information required to achieve the desired network operation. Applet contents will then be transmitted to the server for execution.
- 5) CORBA allows network elements to be given standard Internet addresses so that they may be accessed as any WEB servers.

One might ask, "What does the above have to do with program load?"

Aside from the overall impact that the above scenario will have on future network management systems, it provides very specific possibilities in the area of program load. Consider the following network:

⁷ CORBA, Common Object Request Broker Architecture



Program load based on the above scenario may be implemented in a number of ways - one is described below.

An operator located, in principle, anywhere in the network will log on to a suitable management server via Internet. Aside from any special password and/or additional security measures, the server will behave as any other ordinary Internet WEB server.

After the server is satisfied with the operator's identity and the operator's service request (i.e., a program-load-update), it will download a JAVA script including an applet which will function as the operators input/output interface to the server.

The operator will now be presented with a menu (applet) from which the node or nodes that are to be affected by this program-load-update-request are to be selected (e.g., by highlighting or double-clicking on items in the menu).

The selections will now be sent to the server for processing.

Note that in the above client/server sequence, the operator is the client and the management server is the server. The Java applet executes in the client's machine and then transmits the input to the server for further processing. Any further output to the client may either occur via the first applet that was sent or via a new applet.

The management server is now in possession of a list of node(s) that are to have their software updated.

Since all network elements may now be addressed via Internet (made possible by CORBA), the management server will, acting as a client, log into each node individually and request a software status check, i.e., a comparison of operational versions, release dates, etc. against a "released-software" list.

An applet will be sent from the node to the server for program status verification. The server will execute a JAVA script comparing its list to the software information received in the applet. All differences including information as to the status of new software releases will be sent to the node.

Nodes will now log into a suitable management server and request a program-load-update. They will present their operational program IDs to the server and request the new versions to be downloaded. The server will verify that the new versions do in fact exist and will download them to the node.

The node will now distribute the versions to its local equipment as required.

Finally, the node will report back to the management system that originally ordered the operation, indicating the status of the update.

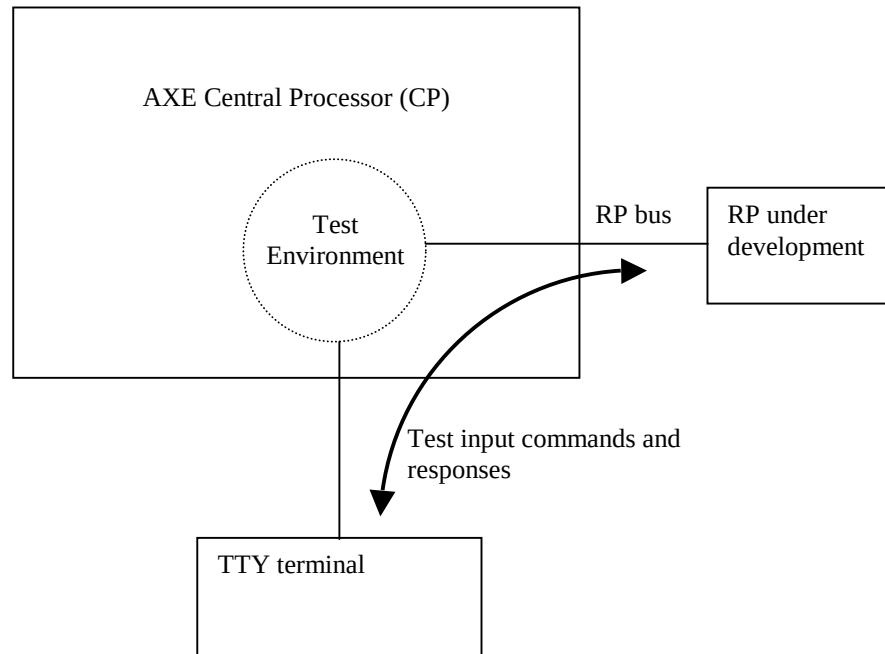
2.2.6 The Razor Development & Testing Environment

The following table provides an overview of the development and testing tools supported by the Razor environment as well as their suitability in supporting the various product phases.

Razor development, debug, test, and troubleshooting tools			
<u>Phase</u>	Hardware	Basic Razor Software	Applications
Development	CAD, VHOL-Simulators, Board Parameters	Diab-C, dmake, UNIX, SCCS, SoftOSE, Obsrec, RPGmap, MSBpe, tip	Diab-C,C++, gmake, ClearCase, SoftOSE, Obsrec, Gnu compiler, ClearMake, UNIX, pde, Signalhandler, RPGmap, MSBpe, tip
Basic Testing	PBOOT, *.pba, P_BIST, JTAG, Monitor	SingleStep, MHP-Linkh, OSEAPI, Illuminator, RPconsole, rpload, DDS (on-board) printd.ose	Same as for basic Razor software test, plus: Sea, rpXsim, purecoverage, qac, qac++ Plus test programs based on MHP and OSEAPI.
Functional Testing		Restart Dumps, Board Parameters, RDS, PC-based SingleStep	Same as for application basic testing.
Manufacturing	PBEM, P_BIST	Restart Dumps	
Maintenance		Restart Dumps	Restart Dumps

2.2.7 Historical background

Prior to 1990, AXE RP product development was generally carried on as illustrated below.

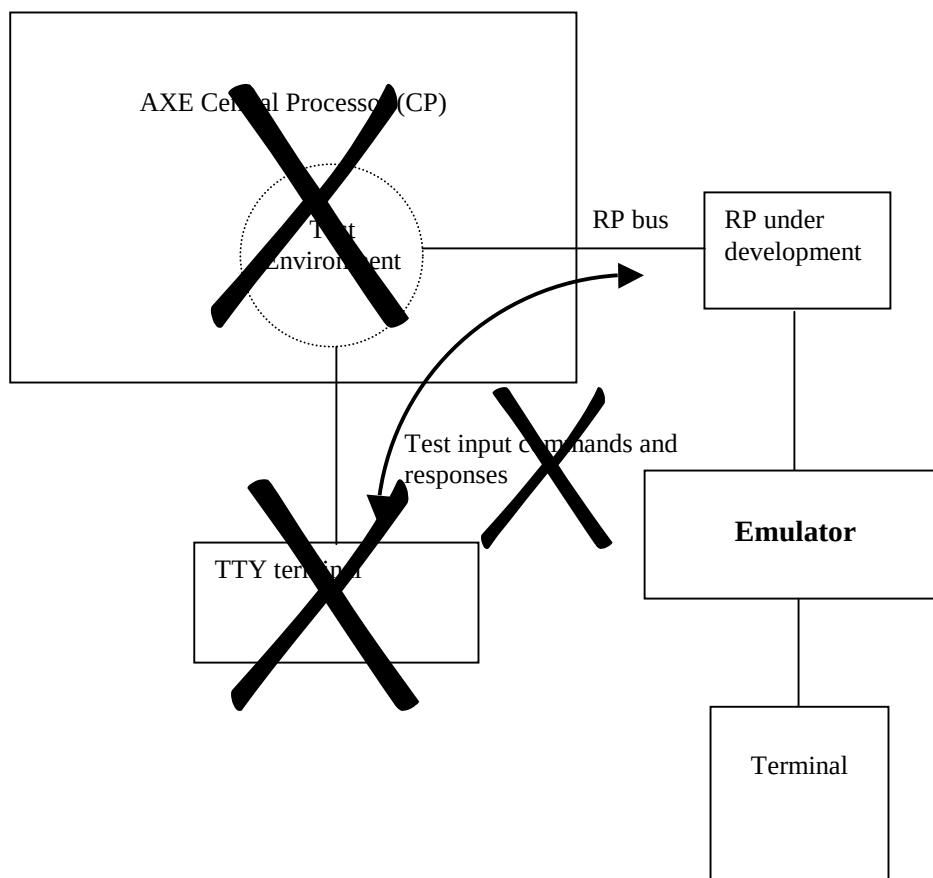


In essence, the development of a new RP product required an AXE system for all test and debug activities.

The entire test environment resided within the AXE central processor and a simple Teletype compatible terminal was used to input commands and to register the responses of the object under test.

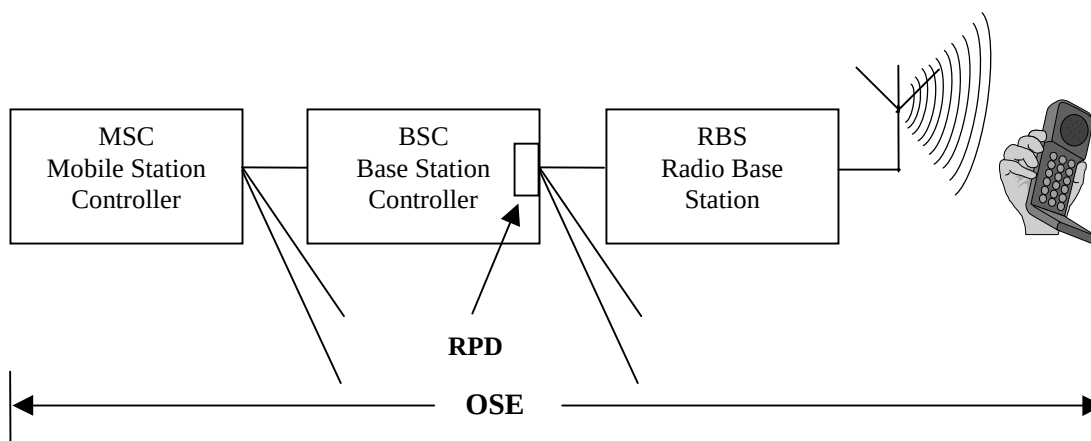
All this changed during 1990 when Ericsson Radio commissioned the development of the RPD for its coming GSM product line.

The new development approach that was to be used is illustrated below.



Programs were now written in C instead of Assembler and the earlier rudimentary operating environments that were tailor-made for each new development project were replaced by Enea's real-time operating system, OSE.

Ericsson's GSM specifications called for OSE use across the board, i.e., OSE was to be used in MSCs, BSCs, RBSs and in Ericsson's mobile telephones.

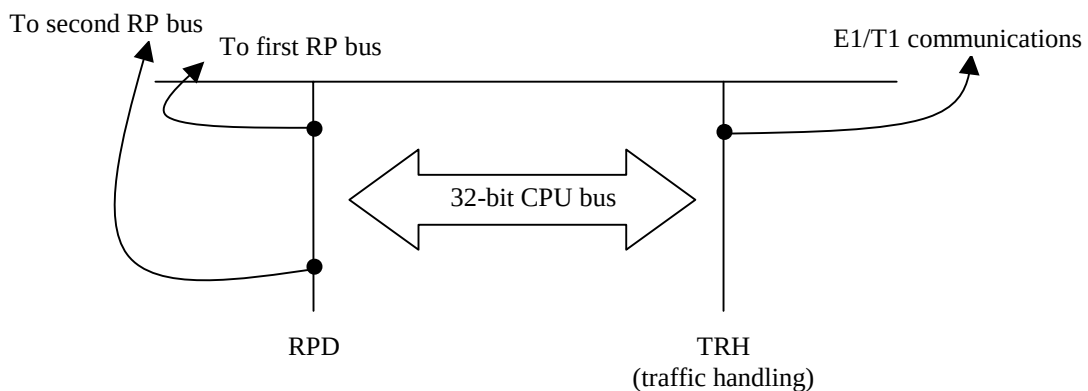


Note that both the MSC and the BSC were AXE-based.

Time to market of Ericsson's GSM line was critical, and the RPD development project was therefore pressed for time.

Why was the RPD required?

- The RPD was needed since the large protocol stacks required by GSM could not suitably be developed in PLEX or in Assembler using earlier RP versions. C was therefore a requirement.
- GSM required the use of multiple boards (see illustration below). Earlier RP designs could not satisfy this requirement.
- OSE, a general and powerful operating environment was required across the entire GSM product line.



The above development approach was more or less successful up to and including basic testing but failed when full-blown functional testing was attempted. Basic testing was performed using a small AXE test environment, however the emulator could not function in the “electrically noisy” conditions under which functional testing was conducted, i.e., when a complete AXE configuration was used.

Test and debug activities were made difficult due to the fact that C application programs executed relocatably, i.e., there was no direct linkage between the errors that occurred and the source code.

Ericsson GSM application development units were severely hampered by the above shortcomings.

It is interesting to note, that dumps that come in today from RPD-based systems cannot always be solved.

This was more or less the situation between 1990 and 1994. In 1994, development was started on the RPG in order to satisfy increased mobile capacity requirements.

The following scenario was to govern the development of the RPG.

1. Flash memory would be used in the design of the RPG board.
2. The emulator was to be done away with. It didn't work in the field.
3. The emulator was to be replaced by a ROM monitor residing in the RPG board which was to be accessible via an RS232 port (TTY A).
4. The board would to contain a second RS232 port (TTY B) that would allow a common on-board protocol (the Link Handler) to be accessed by both applications and the operating system.
5. Debug flags would be implemented as board parameters and be resident in the board's Flash memory.
6. Firmware (*.pba and *.ose) software capabilities would be implemented that would support advanced debugging.
7. Relocatable code was done away with thereby providing C-language debugging capability, i.e., source code debugging.
8. Signal symbol tables were incorporated in all applications leading to improved system-level debugging, i.e., the handling of application signals could now be traced.

3 Glossary

ANSI	American National Standards Institute.
Application	A separately linked OSE program.
Application (Autostarted)	A program that is automatically started by the OS program loader. These types of programs can either be found in the primary or in the secondary file system.
Application (Normal)	A program that is not automatically started. Operation & Maintenance software is generally responsible for starting normal applications.
Autostarted primary file system programs	This type of program is stored in the PBOOT file system and is automatically started by RAZOR. They must therefore be linked to real physical RAM addresses during program generation. *.ose programs are run by both the fallback OS and the normal OS. *.auto programs are only run by the fallback OS. These programs execute in supervisor mode and make use of the system pool.
Block	A group of processes in the OS kernel.
Board Parameter	A read-only non-volatile parameter (stored in the PBOOT flash parameter area) normally contains board configuration information. Indicated by an "*" in front of the parameter name when using the PBOOT CLI.
BSP	Board Support Package.
Cache	Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access ordinary RAM. As data and instructions are processed the microprocessor, the processor first looks in cache memory. If the data or instruction is there (i.e., previously read in from RAM memory), the processor need not bother with the more time-consuming access of the slower RAM memory.
CLI	Command Line Interface.
Cold start	When the board starts, PBOOT is the first program that is started. PBOOT sets up the hardware and copies the programs from PROM to RAM and then starts the operating system. The reason for performing a cold-start can be a power-on or the detection of a fatal error during the execution of an application or a sub-unit of the operating system. Examples of fatal errors are a watchdog timeout or a parity error. A cold start is a complete restart. Since RAM contents are not reliable, RAM is first cleared (initialized to zero) after which files are extracted from the PBOOT file system and loaded into RAM. Very

	little information survives this type of restart.
Console port	The board generally has two console ports: - Port A is used by the PBOOT program and for source code debugging. - Port B is used to connect to the management system built into the operating system S/W.
CPU	Central Processing Unit is another term for processor and microprocessor. The central processing unit of a computer contains the logic circuitry that executes the instructions of the programs stored in the computer's memory.
DDS	Distributed Debug Server.
DHCP	Dynamic Host Configuration Protocol.
DMA	Direct Memory Access is a capability provided by some computer bus architectures that allows data to be sent from an attached device directly into the computer's memory. The processor is free from any involvement in the data transfer - thus speeding up overall computer operation.
DS	Data Store in main memory.
EMC	Electromagnetic Compatibility.
Fallback OS	The read-only version of RAZOR stored in the primary file system during board production. The program is stored with the name BOOTSTRAP. This is the program normally started by the PBOOT program.
Fragmentation	Occurs when the free physical memory of a system is divided into small areas. The problem appears when a new area is to be allocated and it cannot due to the fact that there is no available contiguous area that is large enough.
Host	Workstation or PC
Illuminator	ENE A OSE host program for system level debug. Replaces DDS. Illuminator uses a graphical user interface (GUI).
IP	Internet Protocol.
Kernel	The core of the operating system - provides basic functionality such as process handling and the sending of signals.
LED	Light Emitting Diode.
Link Handler	Handles signal exchange between remote processes in an OSE system.
Local management	The embedded SW is normally managed from a remote management system – however some debug facilities are available locally. This is provided through the local connection of a VT100 terminal (or workstation containing terminal emulation Software).

	Local management allows the contents of the primary and secondary file systems to be changed as part of performing system debug.
Logical address	A logical address is translated by the Memory Manager into a physical address.
MAC	Media Access Controller.
Main pool	Common pool shared between all normal applications.
MMU	Memory Management Unit. An MMU is often part of the processor and provides translation between logical and physical addresses for both memory and I/O accesses. It also provides various levels of access protection.
Page	A page is a group of RAM memory cells that are accessed as part of a single operation. For example the size of a page in a PowerPC 750 is 4-kbyte.
P-BIST	Production Built-In Self Test.
PBOOT	Primary BOOT. The first program to execute after a power-up, watchdog timeout or other hard reset of the CPU. The PBOOT is responsible for: - The initialization of all basic H/W - The management of a simple flash file system - The extraction (decompression) of all files from the file system to RAM - The execution of the file named BOOTSTRAP (RAZOR) - The provision of an API to other programs, especially RAZOR - The provision of a CLI to all users
PDR	Protocol Driver.
Persistent data	Data that survives a warm restart of the application and/or operating system.
Pool	A pool is a memory area from which signal buffers, stack and kernel areas are allocated.
Primary file system or the (PBOOT file system)	A simple flash file system managed by the PBOOT program. Only accessible using the PBOOT CLI. Uses a small part of the flash prom.
Primary file system	An area in the flash memory holding programs and parameters managed by the PBOOT boot program.
PS	Program Store in main memory.
PSTN	Public Switched Telephone Network.
RAM	Random Access Memory is the place in a computer where the operating system, application programs, and data in current use are kept so that they can be quickly accessed by the computer's

processor. RAM is much faster to read from and write to than other forms of computer storage. However, data in RAM only exists as long as the computer is up and running. When the computer is turned off, all RAM data is lost.

When the computer is turned on again, the operating system and other files are loaded into RAM once again (usually from ROM). The most common and least expensive type of RAM is DRAM, dynamic RAM.

RAZOR	OSE Delta with additional S/W for error handling, memory management, program loading and other facilities.
RDS	Remote Debugging System. Commands are the same as in DDS.
Region	A memory area with common characteristics.
REOS	A system call interface between the application and the regional operating system.
Restart	Restarting the system entails stopping current program execution and going to a well-defined location in either the operating system or the boot program. The type of restart can be either cold or warm. Restarts may be ordered via command from the management system. For the most part they are automatic and are caused by an application or operating system fault.
ROM	Read-Only Memory, ROM is computer memory containing data that normally can only be read. ROM contains the necessary data and programs that allow a computer to be "booted up" or regenerated each time it is turned on. Unlike a computer's random access memory (RAM), the data in ROM is not lost when computer power is turned off. PROM, programmable ROM, is a type of ROM that may be written to during runtime.
SDS	Software Development Systems.
Secondary file system	A flash file system used by the operating system itself. The secondary FS uses the major part of the flash prom.
Segment	An address space, which can be independently accessed from an application program using the support in the OS kernel and the CPU. This is not the same as a segment in UNIX operating system.
Shadow RAM	Memory area used internally by the memory manager for accessing its internal data.
Single Step	A high-level source-code debugger. Runs on the host system.
Static Process	Process created by the OSE kernel at system-start. These processes have been configured into the kernel at compile time. It is considered as an error to kill a static process.

Supervisor mode	Privileged execution mode. Execution mode used when the operating system is active. The normal CPU privileged mode for the operating system. All system resources are accessible.
System pool	Pool used by the operating system and all autostarted applications.
Target	The board containing PBOOT, RAZOR and the application S/W.
TCP	Transfer Control Protocol.
TFTP	Trivial File Transfer Protocol.
UDP	User Datagram Protocol.
User mode	Normal execution mode for applications. Restricted access to system resources.
Warm start	Restart within the operating system. The contents of RAM are retained. A warm start is initiated and performed by the error handling function upon detecting a fault that does not require a cold start. RAM contents are reliable and the OS is restarted from its entry point. Certain RAM contents survive which are used to aid debugging (for example the restart dump). Errors normally result in the warm start of the system.
Watchdog (H/W)	A H/W function (timer) that supervises S/W. If the watchdog timer expires, the board is reset (cold start) in the same way as it is following a power-up. It is not possible to store data to indicate the reason for the time out.
Watchdog (S/W)	A program execution watchdog function implemented in software, also referred to as the softdog. When the softdog expires, the board restarts (warm start) and a dump is produced.
Virtual memory	Virtual (or logical) memory is a concept that, when implemented by a computer and its operating system, allows programmers to use a very large range of memory or storage addresses for data and program storage. The computing system maps the programmer's virtual addresses to real hardware storage addresses. Usually, the programmer is free from having to concern himself with the availability of data storage.
VT100	An asynchronous terminal using ASCII characters. VT100 screens have a text-based appearance. Ordinarily workstations and PCs have terminal emulation software (e.g., Kermit or tip) that emulates VT100 functionality.

This page is intentionally left blank

4 Ownership rights and Interpretation

The information in this document is believed to be correct as of the date of publication. This document is however subject to change without notice, and does not represent a commitment on the part of Ericsson.

In the event that any questions arise as to the interpretation of this document, Ericsson will have preferential rights in its interpretation.

This page is intentionally left blank

5 APPENDIX

5.1 The world of real-time

The design of real-time systems is becoming increasingly important. Computers are being incorporated into cars, aircraft, manufacturing assembly lines, and other applications to control processes in real-time, i.e., as events occur. It is not acceptable in such situations to provide input to the computer, allow it to compute for some indefinite length of time, and then examine the output. The computer output must be available in a timely fashion, and the processor must be carefully chosen and the tasks scheduled so that deadlines are met. Frequently, real-time tasks repeat at fixed time intervals; for example, every so many seconds (often milliseconds), sensor data are gathered and analyzed and a control signal is generated. In such cases, scheduling theory is utilized by the systems designer to determine how tasks should be scheduled on a given processor.

Note: The discussion below is based primarily on the OSE DELTA System, a product of ENEA DATA AB.

What exactly is meant by real-time?

The following may be considered to be an orthodox definition of a real-time system:

"A real-time system is one in which the correctness of the processing does not only depend upon the correctness of the data to be processed or on the logical correctness of the various computations that are performed, but also on the time at which the result is produced. If the timing constraints of the system are not met, then system failure is said to have occurred."

It is therefore essential that the timing constraints of the system be guaranteed. Guaranteeing timing behavior requires that the system be predictable. It is also generally desirable that the system attains a high degree of utilization while at the same time satisfying the timing constraints of the application.

An example:

A robot is to pick up a part from a conveyor belt. The part is moving and the robot has a small window of time and space to pick the object up. If the robot is late, the piece won't be there any longer, and thus the job will have been done incorrectly - even though the robot went to the "right place". If the robot is early, the part hasn't arrived as yet, and the robot may disturb the order of the parts moving along the belt.

A Real-Time Operating System (RTOS)

A real-time operating system is a set of programs that must respond to external events within a limited time interval. A real-time operating system is a platform suitable for supporting real-time applications.

A PC user waiting for the results of a compilation to arrive from a network server does not feel that a disaster has occurred if the results arrive ten or twenty

seconds later than expected. On the other hand a delay of ten seconds in a space shuttle application may literally mean the difference between life and death.

As part of its duties related to the management of system resources (i.e., the CPU, memory and various peripheral devices), a real-time operating system must support application programs that require the ability to branch into a number of concurrent processes that may or may not have the same processing priority as the process that created them, the ability for higher-priority processes to interrupt lower-priority processes (pre-emptive), and the ability for processes to synchronize and communicate with one another.

Razor a real-time operating environment includes much more than simply a real-time operating system. It includes a well-developed set of tools for the development, test and tuning of the application.

5.1.1 Some important real-time processing concepts

The process is the most fundamental entity in a real-time system since it is the process to which CPU time is allocated by the RTOS. A number of different process categories exist in order to cater to all possible application needs.

Static and dynamic processes

The static process is the most common process form. Following initiation, a static process will exist “forever”, i.e., for the life of the system or for the life of the software unit of which the process is a part. No process may “kill” (terminate) an existing static process. Static processes may only be terminated if part of a separately loaded software unit and only in conjunction with the termination of that entire software unit.

As opposed to static processes, dynamic processes may be freely created and killed as needed. The dynamic process allows for the execution of the same code by more than one process – note that the number of processes need not be specified at compile-time. Dynamic processes also allow stack-space to be optimally utilized since space is only allocated during the life of the process.

A process is always in a defined state - Running, ready or waiting

A running process is one that is currently assigned to a CPU for processing. By definition, a single CPU system may only have one process in this state at a time.

Processes that are ready to run are placed in a ready-queue. In the event of a process-switch, the first process in the ready queue will be scheduled for execution.

A priority scheme exists in which each priority level has its own ready-queue. Whenever a process-switch is performed, a process will be selected for scheduling from the ready-queue having the highest priority.

A process is said to be “waiting” whenever it is waiting for an event to occur or is temporarily stopped. Processes do not require CPU resources while in this state.

Scheduling principles - Pre-emptive, cyclic, priority-based and round-robin scheduling

Regardless of its priority, any process may be pre-empted (i.e., interrupted by the RTOS) at any time. After which the RTOS may either return control to the pre-empted process or give control to a new process.

Cyclic processes are those that are scheduled to run at certain time intervals. This method is used for timer interrupt processes.

The process having the highest priority will always continue to execute as long as there are no interrupts and the process does not switch to the wait-state.

As mentioned above, every priority level has a ready-queue. Whenever a process is selected for CPU scheduling, it is always the first process in the queue of the highest priority level that is selected. When an active process voluntarily enters the wait-state it will be placed at the end of its present queue (after having once again entered the ready-state) and will wait its turn to be scheduled. A process that is pre-empted by the RTOS due to the occurrence of some higher priority event will retain its place in the ready queue and will be activated again when the priority of its ready-queue is higher than that of any other ready-queue. Background processes are always placed at the lowest priority level. Unlike real-time prioritized processes, background processes are given a specified time slice when scheduled (round robin scheduling). When the background process' time slice expires or in the event that it is pre-empted, the process (after having entered the ready-state) will be placed at the end of the queue and await a new time slice.

Different process types - Interrupt, timer-interrupt, prioritized, background and phantom process-types

Interrupt processes are activated in response to a hardware interrupt or a software event and will run from beginning to end, i.e., they may not be interrupted, unless an interrupt process of higher priority wants to run. An interrupt process that is pre-empted by a higher-priority interrupt process will continue its processing after the higher-priority process has concluded. Interrupt processes always have higher priority than either prioritized processes or background processes.

Timer-interrupt processes behave in the same way as ordinary interrupt processes except that they are activated in response to changes in the system timer.

The prioritized process is the most common form of process type. As mentioned above they run as long as no interrupt process requires CPU resources or are pre-empted by a higher-priority prioritized process.

Background processes are run in round-robin mode at a priority level below prioritized processes. This means that background processes may be pre-empted by both prioritized as well as interrupt processes. The time-slice mechanism is employed to distribute leftover CPU resources to background processes. A background process will not be forced to relinquish CPU control until its time-slice has expired. This means that it may enter the wait-state any number of times and keep control of the CPU as long as its time-slice has not

expired - and naturally, as long as no higher priority task interrupts it. After time-slice expiration, the process will be placed at the end of the queue and await scheduling once again.

Phantom processes do not contain executable code. Generally they only contain redirection tables to redirect signals.

Interprocess communication – signals, signal redirection and semaphores

Three different methods are available for communicating between or synchronizing processes.

Signals (i.e., messages) may be sent from one process to another.

In certain situations, signals must be processed by a process other than the one to which the signal was sent. For this purpose, processes may optionally be equipped with “redirection tables”.

Each process has one “fast semaphore”. Fast semaphores may be sent for inter-process communication and are much faster than ordinary signaling – fast semaphores do not include any data.

Semaphores are generally used to ensure that only one process at a time is using a globally shared resource.

Environment variables

Environment variables are data strings attached to a process. Their primary function is in the configuration of the process that owns the particular variables.

Environment variables may be seen as facilitating a degree of run-time adaptation, i.e., programs can adapt their behavior depending on the setting of these variables.

Interrupt handling – Hardware interrupts, wakeup facility and timer interrupts

Interrupts cause execution to be switched to a separate interrupt process as soon as possible after the receipt of the interrupt.

Interrupts are generally caused by external hardware events, e.g., the reception of an incoming packet on a network interface.

An interrupt process can be activated (or awakened) either by sending a signal to the interrupt process or by signaling the interrupt's fast semaphore. This facility is primarily used in the debugging of device drivers or in conjunction with the creation of device drivers having signal interfaces.

Timer interrupt processes are identical to ordinary interrupt processes except for the manner in which they are called, namely, the number of elapsed milliseconds between process activations is specified at the creation of the timer process.

5.2 Embedded systems

Early embedded systems were based on the use of very simple CPUs. They were a convenient way to replace complex hardware with a simple controller that could perform relatively complex operations. As these operations became more complex, it was worthwhile to separate firmware into system operating software and applications code.

Generally, this partitioning was recognized as being worthwhile, so embedded operating systems survived the laboratory and were released into the wild. The embedded systems those operating systems supported became more complex. As this complexity grew, embedded systems were forced to add support for more and more "computer" things, such as communications channels, storage devices, and display devices.

Traditional embedded operating systems started small and have had to grow with the applications.

Today numerous industrial and commercial products come with computers, microprocessors in particular, inside them. These embedded computer systems can perform many complex functions. Although most products do not look like computers on the outside, they are actually special-purpose computers. For example, air conditioners maintain room temperature with the aid of microcomputers more accurately than without them. Automobiles use embedded computer systems to control fuel injection and other driving mechanisms more efficiently, and wristwatches incorporate alarms, telephone numbers, calendars, and other features. Telephone exchanges are essentially gigantic special-purpose computers that have speech-path connection mechanisms as output units; many complex telephone services are provided by enormous computer programs. Game machines at arcades also are equipped with powerful microprocessors; they use joysticks or other mechanical devices as input units and color monitors and sound generators as output units. Videocassette recorders, telephone sets, cameras, photocopiers, sewing machines, industrial robots, hand tools, military weapons, and medical instruments are other examples of embedded systems.

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 lion's share of the embedded market, the market is experiencing tremendous growth and innovation, with no end in sight.

To get an idea of how widespread these hidden systems are becoming, consider that the market for these devices is seen as growing 30% to 1.2 billion by 2001.

Where is this all leading? Eventually you will hold a gadget in your hand that is as smart as the low-end PC you're using today. And many of the estimated 35 to 50 devices in your home today that contain microprocessors will eventually talk to each other.

5.2.1 Speculating as to the future of real-time embedded systems

The majority of electronic machinery existing around the late 50's and early 60's was "programmed" through hardwiring - all of a machine's functionality was designed into the machine through the use of "hardwired" electronic Boolean logic circuitry.

Not long thereafter, the stored program machine emerged, whereby a machine's electronic circuitry basically was used for the decoding of stored machine instructions that performed various logical and arithmetic manipulations on the data that was fed into the machine.

Real-time computing concepts originally surfaced around the early to mid 60's and were for the most part (during the late 60's and early 70's) applied to the area of so called on-line systems such as airline ticketing systems and later banking systems. This was the same period during which the first database systems saw the light of day - more or less a prerequisite for the operation of the aforementioned on-line administrative systems.

A wealth of knowledge and craftsmanship was generated during the years that followed, specifically in areas such as real-time system configuration, sizing, response-time and queue simulation.

It was during this same period (mid-sixties) that smaller systems (minicomputers) appeared on the market and eventually found their way into OEM systems that were delivered by manufacturers of manufacturing systems, medical life-support systems, laboratory systems, etc. Although very different from today's systems, these may be considered as being the first "embedded" systems.

Computer systems, architecture, software, networking and applications propelled upwards in power, functionality and complexity and somewhere around the early-to-mid eighties system complexity started to get just too complicated for the creation and use of deterministic models – i.e., models that allowed real-time applications to be configured so that stringent response time requirements could be guaranteed with a reasonable degree of certainty.

It was at about this time that chip design started to come into its own and in 1988 the 1-million transistors/chip mark was passed. As a matter of interest, it will not be long before the 50-million transistors/chip level is passed.

The lack of dependable real-time models for the design of real-time software applications caused system designer focus to shift from pure software platforms to a mix of both hardware and software, i.e., real-time system design shifted from being a more or less purely software prospect to applications that were partially "coded" in chip circuitry.

Today, somewhere around 80-95 percent of our embedded system real-time functionality is handled by chip circuitry. The remaining 5-20 percent is handled by an embedded real-time operating system.

Future embedded system architecture may very well look as follows:

