A Comparative Study on Real Time Enhanced Linux Variants

Nicolas McGuire et al.

OpenTech EDV Research GmbH
June 18, 2005

Contents

0.1 Foreword xvii
0.1.1 Goals xvii
0.1.2 Themes xix
0.1.3 List of participants xix
0.1.4 Note on Open-Source xix
0.2 General Purpose Operating System - a brief introductions . . xx
0.3 Basic Architecture of a GPOS xx
0.4 GPOS Extensions xxi
0.4.1 Subsystems and Daemons xxi
0.4.2 User Space xxii
0.5 Functionality of a GPOS xxii
0.5.1 Hardware Abstraction xxiii
0.5.2 Memory Management xxiii
0.5.3 Process Management xxvii
0.5.4 Data Storage xxix
0.5.5 Communication xxx
0.5.6 Networking xxx
0.5.7 Inter Process Communication(IPC) xxxi
0.5.8 Security xxxii
0.5.9 Non-RT Optimization in GNU/Linux xxxiii
0.5.10 User space applications xxxv
0.5.11 User Interface xxxvi
0.6 Guiding Standards xxxvii

I Real Time Linux **1**

1 Introduction **3**
1.1 RTOS 3
1.2 RTOS Design dilema 5
1.2.1 Expand an RTOS 5
1.2.2 Make a General Purpose OS Realtime Capable 6
1.2.3 GPOS vs. RTOS performance 7
1.3 Dual Kernel concept 8

1.3.1 RTLinux Patent 8
1.4 The RT-executive 9
1.5 What happens to Linux 17
1.6 What happens to dynamic resources 18
1.7 Preemtive Kernel 19
1.8 Overview of existing RT-extensions to Linux 19

2 Kernel Space API **21**
2.1 General 21
2.1.1 Thread 22
2.1.2 Timers 22
2.1.3 Interrupts 23
2.1.4 Signals 24
2.2 RTAI (both RTHAL and ADEOS) 24
2.2.1 Non-POSIX Kernel-space API 25
2.2.2 Kernel-space POSIX threads API 27
2.2.3 Signals 29
2.2.4 RTAI BITS - the real signals ? 30
2.2.5 Interrupts 31
2.2.6 Timers 33
2.2.7 Backwars/Forwards Compatibility 36
2.2.8 POSIX synchronisation 37
2.2.9 very non-POSIX sync extensions 38
2.2.10 POSIX protocols supported 38
2.3 RTLinux/GPL 39
2.3.1 Kernel-space threads API 39
2.3.2 POSIX signals 40
2.3.3 Interrupts 41
2.3.4 POSIX timer 42
2.3.5 POSIX synchronisation 43
2.3.6 POSIX protocols supported 45
2.3.7 Backwars/Forwards Compatibility 46
2.4 RTLinux/Pro 46
2.4.1 Kernel-space threads API 47
2.4.2 POSIX synchronisation 48
2.4.3 POSIX protocols supported 50
2.4.4 POSIX options supported 50
2.4.5 Non-portable POSIX extensions 50
2.4.6 Signals 51
2.4.7 Interrupts 52
2.4.8 Timers 53
2.4.9 Backwars/Forwards Compatibility 53
2.5 ADEOS 53
2.5.1 Interrupts 54

2.5.2 ADEOS interrupt processing characteristics 55
2.5.3 Performance 60
2.5.4 ADEOS IPC 60
2.5.5 System events 62
2.5.6 Domain Debuging 63
2.5.7 ADEOS Domain Examples 63

3 Accessing Kernel Resources 65
3.1 kthreads 66
3.1.1 simple example 67
3.2 communicating with rt-threads 69
3.2.1 buddy thread concept 69
3.3 tasklets 73
3.3.1 simple tasklet example 74
3.3.2 scheduling tasklets from rt-context 76
3.3.3 naive rt-allocator 77
3.3.4 Tasklets in RTAI 81
3.4 sharing memory 82
3.4.1 Simple mmap driver 82
3.4.2 Using /dev/mem 86
3.4.3 Using reserved 'raw'-memory 89
3.5 non-standard system calls 90
3.6 Shared waiting queue (Experimental) 92
3.6.1 shq API 92
3.7 Accessing kernel-functions 92

4 RT/Kernel/User-Space Communiction 93
4.1 Standard IPC 93
4.2 Synchronization objects 93
4.2.1 FIFO 95
4.2.2 SHared Memory, SHM 100
4.2.3 ioctl/sysctl 102
4.3 Implementation specific standard IPC 102
4.3.1 RTLinux/GPL message queues 103
4.3.2 RTLinux/GPL POSIX signals 103
4.3.3 RTAI message queues and mailboxes 104
4.3.4 non-standard IPC 107
4.3.5 Performance 112
4.3.6 `/proc/sys` Sysctl Functions via `proc` 114
4.3.7 Security 114
4.4 Interfacing to the realtime subsystem 115
4.4.1 Task control via `/proc` 115
4.4.2 Exporting RT-process-internals via `/proc` 118
4.4.3 Security Issues 120

4.4.4 tasklets . . . 120
4.4.5 dedicated system calls . . . 121
4.5 extended non-standard IPC . . . 122
4.5.1 RTLinux/Pro one-way queues . . . 122

5 User Space Realtime 125
5.1 PSC . . . 127
5.1.1 POSIX signals API . . . 127
5.1.2 User-Space ISR . . . 128
5.1.3 Limitations of PSC . . . 128
5.2 LXRT . . . 129
5.2.1 API Concept . . . 129
5.2.2 Basic concept of LXRT . . . 129
5.2.3 LXRT . . . 130
5.2.4 New LXRT . . . 130
5.2.5 LXRT Modules . . . 130
5.3 PSDD - Process Space Development Domain . . . 131
5.3.1 PSDD API Concept . . . 131
5.3.2 Frame Scheduler . . . 132
5.3.3 controlling the frame-scheduler . . . 133

6 Performance Issues 135
6.1 scheduling implementations . . . 135
6.1.1 RTLinux/GPL scheduler . . . 136
6.1.2 RTLinux/Pro scheduler . . . 138
6.1.3 RTAI scheduler . . . 139
6.2 synchronization . . . 141

7 Resource managment 143
7.1 Dynamic Memory . . . 143
7.1.1 Kernel memory management facilities . . . 143
7.1.2 RTAI memory manager . . . 148
7.1.3 RTLinux/GPL DIDMA (Experimental) . . . 149

8 Hardware access - Driver Issues 151
8.1 synchronization . . . 151
8.1.1 buffering . . . 153
8.1.2 security . . . 154

9 CPU selection Guidelines 155
9.0.3 Introduction . . . 155
9.0.4 RT related hardware issues . . . 155
9.1 Interrupts . . . 156
9.1.1 Shared Interrupts . . . 157

9.1.2 CPM . . . 159
9.1.3 SMIs . . . 159
9.1.4 8254/APIC . . . 159
9.2 Platform specifics . . . 160
9.2.1 ia32 Platforms . . . 160
9.2.2 PowerPC Platforms . . . 161
9.2.3 Platforms known to cause problems . . . 162

10 Debugging **163**
10.1 Code debuging . . . 163
10.1.1 Non-rt kernel . . . 167
10.2 Temporal debuging . . . 167

11 Support **173**
11.0.1 Community support . . . 173
11.0.2 Commercial support . . . 174

12 Reference Projects **175**
12.1 Information sources . . . 175
12.1.1 Variant specific references . . . 176
12.2 Some representative Projects . . . 176
12.2.1 RT-Linux for Adaptive Cardiac Arrhythmia Control . 176
12.2.2 Employing Real-Time Linux in a Test Bench for Rotating Micro Mechanical Devices . . . 177
12.2.3 Remote Data Acquisition and Control System for Mössbauer Spectroscopy Based on RT-Linux . . . 177
12.2.4 RTLinux in CNC machine control . . . 178
12.2.5 Humanoid Robot H7 for Autonomous & Intelligent Software Research . . . 178
12.2.6 Real-time Linux in Chemical Process Control: Some Application Results . . . 179

II Main Stream Linux Preemption **181**

13 Introduction **183**

14 Mainstream Kernel Details **185**
14.1 Time in Mainstream Kernel . . . 185
14.1.1 Current Time . . . 186
14.1.2 Delaying Execution . . . 186
14.1.3 Timers . . . 187
14.2 Scheduler . . . 190
14.2.1 Mainstream Scheduler . . . 190
14.3 High Resolution Timers . . . 192

14.3.1 Overview and History . . . 192
14.3.2 Design and Implementation . . . 192
14.3.3 Summary . . . 195

15 Kernel Preemption in Mainstream Linux 197
15.1 Preemptible Kernel . . . 198
15.1.1 Overview and History . . . 198
15.1.2 Design and Implementation(Modification) Details . . . 199
15.1.3 Some Test Results . . . 201
15.1.4 Summary . . . 202
15.1.5 Notes . . . 203
15.2 Low Latency Option/Patch . . . 203
15.2.1 Overview and History . . . 203
15.2.2 Design and Modification . . . 204
15.2.3 Summary . . . 207
15.2.4 Guidelines . . . 207
15.3 TODO . . . 208

16 Preemptive Linux (Soft)Real-Time Variants 209
16.1 KURT . . . 209
16.1.1 Overview and History . . . 209
16.1.2 Design and technical Details . . . 209
16.1.3 Summary . . . 210
16.2 Montavista Linux . . . 211
16.2.1 Overview and History . . . 211
16.2.2 Design and Technical Details . . . 212
16.2.3 Notes . . . 214
16.3 TimeSys RTOS . . . 215
16.4 Others . . . 216

17 Appendix 217
17.1 Benchmarks . . . 217
17.1.1 Latencies of Linux Scheduler . . . 217
17.1.2 Rhealstone . . . 218
17.1.3 realfeel . . . 219
17.1.4 TimePegs . . . 219
17.2 Trace and Debugging Tools . . . 219
17.2.1 Linux Trace Toolkit . . . 219

18 Webresources 221

19 Glossary 223

III Real Time Networking 227

20 Introduction 229

21 Real-Time Networking 231
21.1 Accessing the Network . . . 231
21.1.1 Direct Arbitration . . . 231
21.1.2 Indirect Arbitration . . . 232
21.2 RTOS Side of the Real-Time Networking . . . 233
21.2.1 Buffering . . . 234
21.2.2 Envelope Assembly/Disassembly . . . 234
21.2.3 Fragmentation . . . 235
21.2.4 Packet Interleaving/Dedicated Networks . . . 235
21.2.5 Error Handling . . . 235
21.2.6 Security . . . 236
21.2.7 Standardizatioan . . . 236
21.2.8 Open Issues . . . 236
21.2.9 CLEANUP:Hardware Related Issues . . . 237
21.2.10 CLEANUP:Non-RT Networking . . . 237

22 Notes on Protocols 239
22.1 RS232/EIA232 . . . 239
22.1.1 Serial Communications . . . 240
22.1.2 Pin Assignments . . . 242
22.2 CAN . . . 244
22.3 IEEE 1394 . . . 246
22.3.1 Topology . . . 247
22.3.2 Physical layer . . . 249
22.3.3 Link Layer . . . 254
22.3.4 Transaction Layer . . . 255
22.3.5 Bus Management Layer . . . 258
22.3.6 1394b . . . 259
22.4 Ethernet . . . 259
22.4.1 Ethernet Network Elements . . . 260
22.4.2 The IEEE 802.3 Logical Relationship to the ISO Reference Model . . . 261
22.4.3 Network Topologies . . . 262
22.4.4 Manchester Encoding . . . 262
22.4.5 The 802.3 MAC Sublayer Protocol . . . 263
22.5 IP (Internet Protocol) . . . 266
22.5.1 IP Addressing . . . 269
22.5.2 Subnetting . . . 271
22.6 Internet Control Protocols . . . 273
22.6.1 The Internet Control Message Protocol (ICMP) . . . 273

22.7 The Transmission Control Protocol (TCP) 274
22.7.1 The TCP Service Model 275
22.7.2 The TCP Protocol 276
22.7.3 The TCP Segment Header 277
22.7.4 TCP Connection Management 280
22.7.5 TCP Transmission Policy 282
22.7.6 TCP Congestion Control 286
22.7.7 TCP Timer Management 288
22.8 The User Data Protocol (UDP) 290

23 Overview of Existing Extensions 291
23.1 rt_com 291
23.1.1 Overview and History 291
23.1.2 Guidelines 291
23.2 spdrv 293
23.2.1 Overview and History 293
23.2.2 Guidelines 294
23.3 RT-CAN 296
23.3.1 Overview and History 296
23.3.2 Guidelines 296
23.4 RTnet 298
23.4.1 Overview and History 298
23.4.2 Guidelines 300
23.5 lwIP for RTLinux 302
23.5.1 Overview and History 302
23.5.2 Guidelines 303
23.6 LNET/RTLinuxPro Ethernet 306
23.6.1 Overview and History 306
23.6.2 Guidelines 307
23.7 LNET/RTLinuxPro 1394 a/b 308
23.7.1 Overview and History 308
23.7.2 Guidelines 310
23.8 REDD/Real Time Ethernet Device Driver 312
23.8.1 Overview and History 312
23.8.2 Guideline 312
23.9 RTsock 313
23.9.1 Overview and History 313
23.9.2 Guideline 314
23.10TimeSys Linux/Net 316
23.10.1 Overview and History 316
23.10.2 Guidelines 317

24 Conclusion **321**
24.1 Hard Real-Time Networking 321
24.1.1 Preference for serial lines: 322
24.1.2 Preference for firewire: 323
24.1.3 Preference for RT-CAN: 323
24.1.4 Usage of ethernet as hard real-time networking infrastructure: 323
24.2 Soft Real-Rime (QoS) Networking 324
24.3 Non Real-Time Connectivity to Real-Time Threads 325
24.3.1 Standard Linux Networking 325
24.3.2 Dedicated non Real-Time Networking 326

25 Resources **327**

IV Overview of embedded Linux resources **331**
25.1 Introduction 333
25.2 The main chalenges in Highend Embedded OS 334
25.2.1 User Interface 334
25.2.2 Network Capabilities 337
25.3 Security Issues 339
25.3.1 Linux Security 339
25.3.2 Talking to devices 340
25.3.3 Kernel Capabilities 341
25.3.4 Network integration 342
25.3.5 Boot loader 343
25.4 Resource Allocation 344
25.4.1 Time 344
25.4.2 Storage 347
25.4.3 Network 350
25.4.4 Filesystem selection 351
25.5 Operational Concepts 359
25.5.1 Available Boot Loaders 359
25.5.2 Networked Systems 364
25.5.3 RAMDISC Systems 365
25.5.4 Flash and Harddisk 368
25.5.5 Linux in the BIOS for X86 373
25.6 Compatibility and Standards Issues 374
25.6.1 POSIX I/II 375
25.6.2 Network Standards 376
25.6.3 Compatibility Issues 376
25.6.4 Software Lifecycle 378
25.7 Engeneering Requirements 378
25.8 Conclusion 380

25.8.1 Borad support packages 382
25.8.2 summary 383

A Terminology **385**

B List of Acronyms **393**

List of Tables

1 Proprietary vs Open Systems (from "Real Time Unix Systems - Design and Application Guide" KAP 1991) xvii

22.1 Minimum data block size 248
22.2 Seventeen signals of the link layer to physical layer interface . 255
22.3 Limits for half-duplex operation 265
22.4 The states used in the TCP connection management finite state machine 283

List of Figures

1 OS diagram - Shell structure of the LINUX GPOS xxi

1.1 classifications for realtime systems 5
1.2 Dual Kernel Concept ([32]) 9

13.1 Kernel Modification Variants 184

15.1 Softrealtime Concepts 198
15.2 Histogram of Latencies [**?**] 202

22.1 Asynchronous serial data frame (8E1) 241
22.2 EIA232 signal definition for the DTE device 242
22.3 EIA232 signal definition for the DCE device 243
22.4 Conventional usage of signal names 244
22.5 A firewire bus 247
22.6 IEEE-1394 protocol layers 249
22.7 Data strobe encoding 250
22.8 Bus after leaf node identification 251
22.9 Bus after tree identification is complete 252
22.10 Asynchronous packet format 256
22.11 A split transaction 257
22.12 Ethernet's logical relationship to the ISO reference model . . 261
22.13 (a) Binary encoding (b) Manchester encoding 263
22.14 The 802.3 frame format 263
22.15 Collision detection can take as long as 2T 265
22.16 The IP (Internet Protocol) header 267
22.17 IP address formats 270
22.18 Subnet address hierarchy 272
22.19 Subnetting reduces the routing requirements of the Internet . 273
22.20 The TCP header 278
22.21 The pseudoheader included in the TCP checksum 280
22.22 (a) TCP connection establishment in the normal case (b) Call collision 281
22.23 Window management in TCP 284
22.24 Silly window syndrome 285

22.25(a) Probability density of acknowledgement arrival times in the data link layer (b) Probability density of acknowledgement arrival times for TCP 288
22.26The UDP header 290

23.1 Internal structure of RTnet 299

0.1 Foreword

The intent of this document is to allow for a reasonably quick and yet reliable decision on what RT (Real Time), extension, if any at all, to the GNU/Linux operating systems is best suited for a specific problem. In surprisingly many cases this decision will result in plain main stream Linux being the best decision so a clear focus is on the capabilities of main stream Linux. To allow this decision to be on the grounds of sound understanding of the key issues an introduction to the core problems of RTOS (Real Time Operating System) implementations in GPOS (General Purpose Operating System) is given preceded by definitions of some of the key terminology used.

This first version of the study is the dry-run version, it is exclusively based on published material and studying of the documentation of the different variants - in this sense it is a preparation for the anticipated second phase study that will include testing and comparing on specified platforms. Nevertheless we believe this study can provide a first level of guidance for managers and engineers that need to make an GPOS/RTOS decision.

0.1.1 Goals

When Linux started in 1991 there was no trace of real-time for Linux to be found and probably not much thought was given to this issue - at the same time work groups on Real Time UNIX were working on concepts and implementations of Real Time enhanced UNIX. As one such effort REAL/IX was started, based on a main stream UNIX (AT&T Sys V) its developing team published there work on REAL/IX [1] in 1991 noting some key issues for the success of a real time UNIX.

	Proprietary	Vs. Open Systems
Advantage for Users	Proprietary System	Open System
Software portability	Months/Years	Hours/Weeks
Database Conversion	Years	Hours/Days
Programmer retraining and Availability	Big Issues	Negligible
Flow of Enhancements	Controlled by Computer Manufacturer	Free Market for Major Innovations

Table 1: Proprietary vs Open Systems (from "Real Time Unix Systems - Design and Application Guide" KAP 1991)

When this was published the authors hardly were thinking of a fully open source real time enhanced GNU/Linux system that would prove the validity of these assumptions several years later. In the light of the above the goals of this study are:

- Introduce technological concepts and terminology of the different Real Time Linux variants as well as the kernel preemption capabilities evolving in the 2.5.X/2.6.X series of Linux kernels

- Ease access to further documents and information on Real Time and GNU/Linux by summarizing available documents and providing extensive web references.

- provide GNU/Linux technological summary information for engineers and managers to allow a selection of

  - RTOS / Kernel Version
  - RT-Network solution
  - RT suitable CPU

  best suited for a given problem under the premise that Open Source Technology is anticipated.

- Summarize available RT related resources and projects as well as present representative sample projects to describe the capabilities of Linux RT extensions on a project basis and not only on a fact-sheet basis.

- Give a reasonably complete overview of technologies associated with embedded GNU/Linux systems to allow a judgement of

  - In house efforts on training
  - available open-source technologies available
  - capabilities and benefits/drawbacks of vendor Dev-Kit solutions
  - support (community and commercial)
  - how colaboration with the open-source community could be implemented.

  for an embedded Linux based project.

- Identify the critical reasons of research and the key questions that need a more in depth answer than what available informations can provide.

- Develop a basic concept for Introducing Embedded GNU/Linux based RT technologies on a company scope.

- Constitute a basis for the continuation in Work Package 4-N

0.1.2 Themes

This study is aimed at GNU/Linux systems in general and Real Time enhanced systems specifically. The history of GNU/Linux is fairly well known, but the underlying mechanisms of interacting with the community of developers that make open source happen sometimes is not so clear to managers and engineers - a main theme of this study is to guide open-source newcomers into this developing paradigm.

A side-theme of this study is to hopefully evolve the big-picture of how the components of GNU/Linux, from the tool-chain, user-land all the way to the Kernel, fit together to produce what is commonly referred to as embedded Linux.

0.1.3 List of participants

Florian Bruckner
Matthias Gorjup
Nicholas Mc Guire
Andreas Platschek
Georg Schisser
Quingou Zhou

0.1.4 Note on Open-Source

It is the understanding of the study team that results and documentation are intentionally to be made available to the open source community at a time and in a form considered suitable by Siemens AG; it is the hope of the participants that this work will be made available to the public in the form best suited to support the open source community, as a first step this still somewhat preliminary version is being released for the Lanzhou Summer School at the Distributed Systems Lab June/July 2005.

0.2 General Purpose Operating System - a brief introductions

An operating system provides an abstraction layer between the platforms hardware and application programs using a well defined interface between a user's program space, kernel space drivers and the underlying hardware. The operating system is a management instance and intentionally is transparent to the users' requests.

To the directly visible management tasks one can count the execution of user programs virtually allowing concurrent execution of multiple programs simultaneously . In this section we describe the architecture and functionality of an operating system and optimization strategies employed in General Purpose Operating Systems (GPOS). Discussion of optimization strategies will be limited to those mechanisms in Linux that are distinct non-realtime optimizations - introducing the difference between realtime and non-realtime OS on a phenomenological ground.

0.3 Basic Architecture of a GPOS

The interest of the user sitting in front of a computer is to use some specific service of the system. From this perspective neither the hardware nor the specifics of how to access this is of interest to the user. In this sense an operating system is functionally simply an abstraction layer. More specifically the lowest abstraction layer in a computer system - the one that directly communicates with the hardware. This core of the the operating system is refereed to as kernel. The name kernel follows from the analogy to a nut, where the kernel is the very heart of the nut surrounded by the nut-shell. In the computing domain, the kernel is the very heart of the operating system. This kernel is surrounded by software layers, provide user authorization and interaction facilities (shells, Window managers, application environments like OpenOffice.org).

An operating system viewed as an abstraction layer provides a general, standardized interface to the underlaying implementation details of the kernel and abstracts the specifics of the computer platform, allowing to run the same program on different operating systems and hardware. Figure 1 shows the shell structure of a standard Linux OS.

The intent of such extensive abstraction layer buildup is to provide one of the key features of UNIX-like operating systems - portability of user-space applications. Although most of this introduction will apply to most UNIX flavours around, some of the details noted are Linux specific and might not apply to other UNIX flavours. For a good introduction to operating systems we refere you to [**?**],[36].

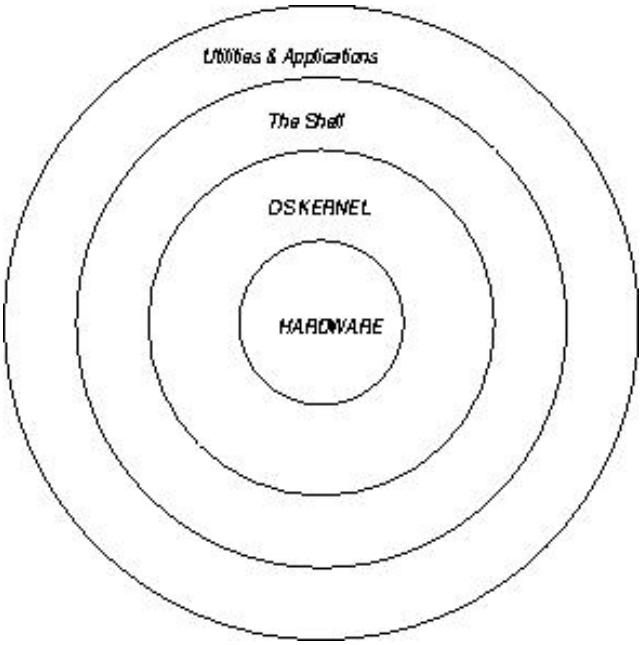

Figure 1: OS diagram - Shell structure of the LINUX GPOS

0.4 GPOS Extensions

A Linux based operating system is generally larger than the kernel proper. Core services of the operating system, that extend the kernel functionality, and are often in user-space for historic reasons, are also resident in physical memory allocated to user space processes, i.e user-space NFS server or the X-Server.

0.4.1 Subsystems and Daemons

GPOS extensions, commonly refereed to as subsystems or daemons, include system logging (klogd/syslogd), timed batch execution (crond/atd), Remote Procedure Call (RPC) (portmap,rpc.mountd,etc.) and Network File System (NFS) (user-space nfsd,r kernel-space knfsd.o). As noted with knfsd (Kernel Network FileSystem Daemon) Linux permits extensions to the Kernel mechanism via loadable modules to extend the kernel functionality in kernel-space. The Linux kernel provides means for automatic loading and unloading in the 2.5.X/2.6.X series, up to only 2.4.X automatic loading via calls to user-space commands as well as loading by directly invoking these user-space commands (insmod/modprobe) was supported, but unloading required manual intervention, in the 2.6.X kernel series automatic unloading is permited which is relevant for resource conservation on embedded systems. A major benefit of loadable modules is that development is simplified:

- changes are local to modules - no recompiling the kernel
- testing can be done without rebooting the system to load the new functionality
- better code isolation - simplifies debugging
- crash cause detection simplified - the problem is well located if the system crashes after inserting the new module
- a further advantage of modules especially for embedded systems is the reduction of the kernel size for fast booting
- generally modules allow updates of core OS features without the need to exchange the entire kernel - which simplifies things for vendors.

Furthermore, relevant in low-resource embedded systems, the reduction of memory used by the kernel by only having those modules loaded that are currently in use if modularized kernels are built (note that 2.5.X and 2.6.X kernels support automated unloading of idle modules).

0.4.2 User Space

The user-space side of the operating system includes high-level abstraction layers like shells, graphical user-interfaces, as well as libraries to abstract resource access in a standardized way. The Linux Operating system is often (and more correctly) referred to as the GNU/Linux OS as its distributions commonly include additional applications such as file management programs, browsers, office suits, code development environments (compilers, debuggers, profiler, etc.), as well as the typical user-space utilities for electronic mail and internet access. User-space programs reside on disk until needed.

The architecture of an operating system is summarized here, somewhat imprecise, as a core (the kernel) that remains in memory during the entire system uptime, a set of processes in user-space that extend the kernel, and a variety of user-space applications and utility programs that remain stored on disk, dynamically loaded by the kernel when requested by users. The kernel manages simultaneous execution of multiple user programs and isolates user programs from the specifics of hardware management and the underlaying hardware platform.

0.5 Functionality of a GPOS

The main functions a general purpose operating system needs to provide are

- hardware abstraction and interfacing

- memory management,
- process management,
- management of persistent data
- communication
- security
- performance optimization

0.5.1 Hardware Abstraction

In very early operating systems the user-programs would directly talk to the hardware, requiring the users to program appropriate sequences to control the hardware, not very user-friendly... The kernel introduces a set of logical devices which allow the user/applications to talk to these logical devices with well defined interfaces in a hardware independent manner. This means your telnet client need not to know that you are using a eepro100 ethernet adapter, it in fact need not even know you are using ethernet. The means by which this abstraction is achieved is that the kernel maintains a set of logical devices available via device-files in most cases (the network devices are an exception) and the low level device drivers use a well-defined interface to the kernel to communicate with users via logical devices. The device drivers register their services with the kernel and the kernel then can use high-level interfaces, like sockets or system calls to pass on user-data to the hardware specific routines in the device-driver.

Fig device driver

As the kernel implements a flat memory model, one can directly access any hardware using the driver functions whose symbols (names) are exported. This allows very efficient hardware access from within kernel space as one does not cross any abstraction layer, but puts the burden of synchronization and proper access to the hardware resources on the programmer.

There are two major ways of synchronizing kernel processes with hardware activity, polling mode and interrupt driven access. The terms are commonly used as synonymes for synchronous/asynchronous access.

- polling - or synchronous access
- interrupt - or asynchronous access

0.5.2 Memory Management

Memory management can be split into three main areas:

- memory allocation

- memory protection
- memory mapping and sharing

The two basic possibilities of memory addressing available are, flat memory model using address registers that are wide enough to address any word in the largest conceivable memory space or a segmented address model, that uses two address registers, one that holds addresses for a block of the memory and a second register that selects the memory location within this block. Which of these strategies is used depends on the hardware of the memory management unit (MMU), X86 and compatibles offer segmented memory, PowerPC and mk68000 family processors use a flat memory scheme. The Linux kernel though does not use the hardware support for segmented memory in any architecture, from the hardware register handling perspective a flat memory model is implemented, the isolation is conceptually done in software (with hardware support by the MMU if available).

The physical memory in the computer is not directly addressed (except for the VM-subsystem [20], and a few other rare exceptions) but is abstracted to a virtual memory of 4GB (default kernel configuration on 32bit systems), reserving one giga-byte in this virtual address space for the kernel (addresses above 0xC0000000) and assigning the lower 3GB to user-space processes. The Virtual Memory subsystem allows the kernel code to be written free of architectural details with respect to memory management, which is why most (more than 95%) of the Linux kernel source is in high-level C-language. Each user-space process also has a virtual memory realm of 4GB of which 3GB are usable (addresses above 0xC0000000 - kernel space is not accessible via the address range), but address spaces of distinct user-space processes are not related unless explicid sharing of memory is done. This means that a user-space process in Linux can allocate more memory than physically available RAM in the system. Linux' VM (Virtual-Memory subsystem) is responsible for mapping a free region of physical memory to the virtual memory being accessed, if necessary moving in-use memory to secondary memory (swap-device) to free up physical memory. Linux' VM views memory not as continuous address range but manages the physical memory as pages of 4kByte (8kByte on 64bit architecutres).

The Memory management is the mechanism provided by the operating system to allocate memory - requested by a process and deallocating memory when a process terminates. Another requirement is to ensure that memory, previously allocated and now no longer required by the process, is released and made available for allocation to other processes when a process exits. This last requirement is known as garbage collection. Note thoug that leveing this to the memory subsystem is conceptually inefficient as an application generally can free memory at an earlier time, that is befor exiting, but the operating system can't forcefully free application memory until the

application terminates. Although it is not a programming error not to free memory in a Linux application, it is inefficient.

The physical media for accessing memory locations in modern processors is not primarily the RAM chip, or primary memory, but is extended for performance reasons to cache memory, that is accessible between 10 to 100 times faster than RAM (L2 cache 10x, L1 cache 100x) but is generally limited to at most a few hundred kilobyte (excluding beautiful architectures like the Alpha that supported a few mega-byte of second level cache...). Naturally these numbers will change with time, but the relation of sizes between RAM and cache can be expected to stay close to what we see in current systems (roughly L2 between 1% and i

As noted above, Linux allows over-committing memory so one needs a means of storing memory on secondary storage media, like a hard-disk, to make enough memory available. For this purpose Linux supports swap partitions on hard-drives, which are a factor 100 to 1000 slower than RAM. A further mechanism is to allow to simply through out read-only aswell as clean-pages (those that were not yet modified) from a process, that then later must be reloaded. The Linux virtual memory subsystem abstracts this hardware-layering completely so that the user has a flat memory of 3GB available for each user-space process.

Memory Allocation

Any multitasking system must be able to decide what physical address should be used by which process, these decisions are taken by the memory allocation code in the kernel, user-space processes switch to kernel mode with a `brk` system call to request memory from the virtual memory (VM) subsystem of the OS. Linux uses a paging system based on virtual memory which provides a flat 32bit address space (4GB) to the applications.

Memory Protection

A basic requirement for a general-purpose operating system is to guarantee data-integrity to every user-space process, this means that every process' address space must be isolated from other user-space processes. No user process should be able to write into the memory location of another process or the kernel, nor should a user-space process have any means of directly accessing physical RAM.

Memory protection is based on the translation of virtual addresses into physical addresses with the kernel assigning the actual physical RAM to each process. This address translation can be done in software or in hardware, generally the software solutions are fairly expensive in terms of CPU usage. As quite a few embedded processors are MMU-less, reducing silicon complexity and thus expenses, GNU/Linux variants for these processors have

evolved quite early (i.e. uClinux - a derivative of Linux 2.0 kernel with only limited multitasking capabilities). For all MMU-less systems supported by the main-stream kernel, user-space memory isolation is preformed in software (i.e. for m68k this is initialized in arch/m68k/kernel/head.S and continued in arch/m68k/mm/motorola.c). These systems implement the same hirarchical page-table based virtual memory sheme as found on systems with MMU. On all platforms supported by GNU/Linux, that do provide a MMU, this is used to enforce memory protection.

The flat memory model is only used in kernel space, that is all kernel space processes share a common memory space including the kernel mode realtime extensions. The underlying assumption is that the experienced programmers writing the kernel code know what they are doing and will not write into memory areas not assigned to the process. This "trusted-code-concept" can be considered valid for the stable series of Linux kernels, as released on `ftp.kernel.org`. Although a memory protection extension to realtime Linux has been demonstrated, all of the main realtime Linux extensions to kernel-space don't provide memory protection, memory protection is enforced though in the user-space realtime extensions.

It should be noted that memory protection is of limited use to a hard real-time system as a faulting hard real-time task would result in missed deadlines even if it does not take down the entire system. Furthermore there is a overhead introduced by memory protection mechanism that increase context switch times, thus increasing the response times of hard real-time systems. Even if this opinion contradicts a majority of publications we concider memory protection a valuable feature during development of real-time systems, but not a crucial issue for runtime systems. The main issue that leads to this conclusion is the dificulty of designing exit strategies for a task that would exit due to a memory violation (segfault), without such exit strategies that maintain real-time constraints memory protection does not prevent system failure.

Memory Mapping and Sharing

Memory not only is used to store data produced by precesses but can also be used by hardware devices like network cards or data acquisition cards to make data available to the OS. This data could now be copied from the physical address where it was deposited (i.e. via DMA) to the address space of the user-space application that processes this data. As data copying is a performance issue, this would though require copying large amounts of data that are actually allready in memory and thus waste performance. As in a virtual memory system any physical memory can be maped into the memory-map of any process this copying can be prevented, simply giving the user-space application direct access to the memory location where data was dropped to. Aside from this form of remapping memory there is a sec-

ond reason for remapping memory, de facto making it available under two distinct addresses, and that is that hardware addresses like the PCI configuration space would make drivers platform dependent. Referencing physical addresses in code breaks the abstraction concept of the virtual memory setup in a GPOS, by remapping physical, hardware specific memory, to virtual addresses and providing an appropriate API to perform this remapping drivers can be written independent of any underlaying physical memory layout. A GPOS must provide this form of memory-mapping to allow platform independent coding of hardware drivers.

Closely related to this is the issue of memory sharing, as every process has its own virtual memory are a direct communication, i.e. via pointers, is not possible simply because there is no relation between the address maps of different processes. Sharing memory means nothing else but creating precisely this relation between two or more distinct virtual memory layouts. The GPOS does this by mapping a given physical address into the memory map of multiple processes and at the same time locking the memory as to prevent it from being invalidated as long as any of the sharing processes is still referencing it. Sharing memory is not only possible between user-space processes (via sysV SHM, and /dev/mem) but also between kernel-space (including rt-context) and user-space and between hardware related memory and user-space applications (i.e. video-memory and X-server). The ability to share memory is at the core of zero-copy interfaces, as the common address space allows to reduce information copying to parsing of the location (pointer) of the information between two or more processes.

0.5.3 Process Management

The Linux kernel has two groups of processes to manage

- kernel processes
- user-space processes

Generally when talking about scheduling we are talking about user-space processes. Kernel space processes like kernel threads, tasklets and interrupt service routines naturallly have a very Linux specific implementation and will be noted in later sections as far as they relate to rt-issues. In the discussion here we will exclude the kernel-space processes for now as they are nither a generally available processing concept nor are the used abstraction concepts generic. It should be noted though that the terminology (i.e. kernel threads) is used in many other GPOS that provide "similar" mechanisms but one should not attempt to transpose findings related to kernel level processes onto other OS.

In a multiuser/multitasking system like GNU/Linux all applications are seemingly running in parallel. This multiplexing of tasks on to a single

CPU is managed by the scheduler. There are two methods that can lead to a user-space task-switch:

- the process relinquishes the CPU voluntarily
- the process is preempted

Linux permits both methods and is thus called a preemptive multitasking system.Note that the term preemtive Os does not refere to kernel level processing (even with the latest preeemptive-kernel patches the kernel is not fully-preemtive, but only permits preemption in particular kernel-states).

The first case where a process voluntarily relinquishes the CPU, by exiting or making a sleep/schedule/etc. system call, returns control to the scheduler, the scheduler selects a new runnable process to execute and switches to that task. The second case, preemption, can have a number of reasons, basically the scheduler is called on some event (i.e. timer interrupt) and selects the highest priority runnable task from the task-list. If a lower priority task had been running then one says that the higher priority process preempted this process. As the task that is preemted needs to continue at a later point in time, the execution context must be saved.

A POSIX compliant scheduler offers three different scheduling policies for processes.

- SCHED_FIFO: Realtime process - the highest priority task is always selected to run, if there are multiple processes with equal priority then the first one in the list runs to completion then the second and so forth.
- SCHED_RR: Real-time process - again the highest priority task is selected but if there are multiple tasks at this priority then the next invocation of the scheduler will select the next task, effectively this is a round-robin selection scheme for tasks of equal priority.
- SCHED_OTHER: Non-realtime process - the effective priority of each task is dynamic - the highest priority process is selected to run. Strictly speaking SCHED_OTHER may implement any strategy it likes, POSIX does not restrict it in any way other than not behaving like SCHED_FIFO or SCHED_RR.

If the only criteria of a process preemption were a fixed priority for all processes in the system then a process could monopolize the CPU entirely and all low priority processes would have to wait for this process to complete. This would destroy the illusion of parallel execution and would be quite inadequate for a multitasking system. The Linux scheduler treats the processes with policy SCHED_OTHER differently, it uses the counter field in the task-structure, a value derived from the static process' priority to

describe the dynamic priority. Before using this value it is tuned. Some of the criteria used are:

- giving a process on the same CPU an advantage
- preferring processes that have the same memory-map

which helps minimize the penalty of context switching (there is quite a bit more of heuristics in the actuall scheduler code `kernel/sched.c`). This dynamic priority is reset after a task actually got a chance to run for a while (its time slice), and increases the longer a task has to wait for running. As Linux is a GPOS and the realtime scheduling policies may not monopolize the CPU, the dynamic priority of a process with polity SCHED_OTHER will eventually become higher than that of any realtime process, thus no process starves, it just runs slower. This method of scheduling is called fair scheduling, one of the prime concepts that makes Linux a non-realtime OS.

The last issue for scheduling is: "what happens when there is no process ready to run ?", in this case the idle task is run. The idle task in Linux can not be killed, so there always is a runnable task on the system. A CPU can't do nothing, it atleast has to be executing a no-op instruction.

0.5.4 Data Storage

Data storage in UNIX like OS is managed via block-devices, these devices don't have access to individual bytes of data but to data-blocks (512bytes to a few kBytes typically), this is one of the rare cases where hardware specific optimization strategies are cast in a file-type in UNIX, generally the file-objects are kept hardware independent (block-devices and char-devices being an exception).

Except for minimum systems (see POSIX minimum system profile PSE 51) basically any OS provides some form of persistent block-oriented storage. UNIX is a file-based GPOS concept, viewing streams and files as two 'states' of data, files being 'frozen' streams. Files or chunks 'frozen-streams' can be stored in volatile (i.e. RAM) or non-volatile media, the GPOS provides the necessary abstraction for the application programmer not to distinguish between the two during coding (open/write to a ram-disk or to a file on hard-disk is not different from the application code). The UNIX way of abstract data storage is thus a storage concept that embeds implicit constraints in the application code (on the contrary MS-DOS had this information explicitly in the file-name `a:whatever` or `c:whatever`). This UNIX way of treating files as totally hardware independent allows for a number of optimization strategies, like caching of files in memory temporarily, preloading of multiple consecutive blocks or read-access, etc. but it requires the application programmer to be aware of these capabilities as it may otherwise lead to

side-effects (i.e. data-loss on files not opened with O_SYNC as the buffered data in RAM may not be flushed on power-failure).

The second picture of persistent data-storage is from the hardware perspective, that is from the perspective of the underlaying storage devices (hard-drive, CF, Flash, floppy). This seconds view has been neglected, attempts to unify hardware layers, like SCSI, or IDE, are specific only to a class of storage hardware (although both have been 'misused' for things like CF-discs). The UNIX file-approach allows a simple abstraction layer 'everything-is-a-file' and the underlaying block-device is of no concern to the file-access methods.

0.5.5 Communication

Any GPOS that should allow more than one process to execute, which by definition is a demand of a GPOS, needs methods to communicate between processes. Communication has two basic purposes:

- data exchange
- synchronization

These two goals of communication methods may be coupled (i.e. signals and sig-info, or UNIX-pipes) or decoupled (i.e. shared memory for data exchanged, and semaphores for synchronization). Having this split available allows application programmers to design very application specific communication layers by combining data exchange and synchronization objects provided by the GPOS.

0.5.6 Networking

Networking is actually a high-level services, due to its high processing demand and its complexity it was integrated as a kernel-service into very early UNIX versions by design. Networking in UNIX is stream oriented (represented in the socket API) and did not maintain the file concept for networking (maybe also a reason why network devices never showed up in the FileSystem - so the device 'eth0'was not represented by `/dev/eth0` as one might expect but rather was an kernel internal object and a specialized API with a set of system calls to access these objects). Networking implementations are an example of a communication layer implemented in kernel-space due to performance **and security issues. Building application or protocol specific kernel-services is an option that an open-source system like GNU/Linux offers to application designers ??, although in most cases this is only feasible for very large projects.**

A further concept in networking that can be utilized in other applications is the layering, placing parts of the layer in kernel-space and others in user-space (i.e. pptp-protocol in user-space

and the underlaying isdn device in kernel-space), the impact on performance of such basic design decision is very high, and needs to be considered very early in a project design state if a dedicated networking layer is being considered.

Generally a GPOS will follow standardized communication protocols like IPv4/IPv6 and provide an appropriate API in library and system calls, this split again is very performance critical.

0.5.7 Inter Process Communication(IPC)

Inter Process Communication (IPC) can be split into communication between user-space applications and the kernel and communication between user-space processes.

Basic IPC mechanisms available in linux for communication between processes ("clasic IPC"):

- **semaphors - semaphors are shared objects used for protection of critical sections (mutual exclusion on access of shared data) (`man ipc`).**
- **shm - shared memory is simply a shared pool of data (pages accessible by more than on e process) - no data transport mechanism is involved (`man ipc`).**
- **fifos/pipes - First in First out unformated (raw) data passwd between processes (`man mknod`)**
- **message queues - similar to fifos just that data is put in envelopes with meta information (message ID and message size) (`man ipc`).**
- **sockets - sockets are bound to network addresses instead of processes but other wise can be seen as similar to message queues (`man socket`).**

Note that the posix threads API provides a further set of IPC mechanisms for multithreaded applications, the list above pretains to processes, in literature the term IPC is sometimes not strictly used only for processes.

Requesting kernel services from within user-space programs is achieved through system calls. These system calls allow users to request access to shared physical (disk, memory, sound card) or logical resources (semaphore, wait queue, network device). In Unix systems, and in Linux, physical resources are accessed from user programs through a FileSystem using POSIX system calls (i.e. `open()`, `close()`, `read()`, `write()`). An exception to this rule,

for historic reasons, are network devices that are accessed via the socket-system calls, and have no representation of the network device in the filesystem associated with them (i.e. no `/dev/eth0`). All of the input/output activity is controlled by the kernel code so that the user-space programs do not have to be concerned with the details of sharing common physical resources.

Inter Process Communication is the class of communication between two processes, between two tasks, or between a process and a task when both are running on the same computer. This is generally referred to as inter-process communication. Inter-process communication is supported by the operating system through primitives such as shared memory, binary signals and pipes. Since two or more processes could be accessing the shared memory segment at the same time, there is a need to indicate when one process is writing to the segment so that the other process(es) wait until the write is complete before performing their own read/write action. This indication is achieved by semaphores whose functionality includes the action of continuing a process that has been placed on hold waiting for the shared resource to be accessible safly again. Pipes are an alternative to shared-memory communication and use different system calls to the shared-memory interface. Shared-memory and pipes allow finite-size messages to be passed between processes while binary signals convey only one bit of information. Signals are another communication mechanism between processes. Only one bit of information is involved, but the operating system may support about 100 such signals with each conveying their own explicit meaning to the receiving process, in addition an OS may support passing information along with signals (siginfo struct) though functionally the signal interface is not intended for data communication. All these inter-process communication mechanisms involve a timing overhead. Signals are faster than pipes which are faster, in turn, than shared memory; sockets are the slowest of all.

0.5.8 Security

Two developments in the past in the embedded world have changed the security demands of embedded systems considerably. First embedded systems have evolved from dedicated hardware to reduced PC-like systems in many cases, utilizing the commodity component computers hardware range, and second a tendency to integrate distributed realtime systems into existing LAN/WANs. These two developments have moved the embedded OS and RTOS from minimalistic OSs focused on a small number of tasks to re-

duced general purpose OS setups, well documented by the many embedded Linux distributions evolving. These developments not only open many new possibilities to the application designer and control engineer but also substantially change the security demands of embedded systems.

Aside from the resource demands that limited security mechanisms applicable to resource-constraint systems, system capabilities are moving towards a reduced general purpose OS mandating strategies to satisfy the security demands of networked systems in general and the specifics of embedded systems in particular. Linux has developed the necessary capabilities on the kernel level (kernel capabilities, encrypted FileSystem, process accounting and monitoring), as well as in user-space (AES-encryption, ssl, virtual-hosts, etc.) . Aside from these efforts to target specific problems, a global security approach was taken with the Security-Enhanced Linux (http://www.nsa.gov/selinux/)

0.5.9 Non-RT Optimization in GNU/Linux

This list is not exhaustive - but it should make clear why it is desirable to have as much of the processing done in a non-realtime environment and limit realtime context to the execution of critical routines only. The optimization strategies noted here are all characterized by the improvement of the average case at the expense of performing fairly slowly in the rare worst case.

- **Dynamic memory: non-rt systems can allocate resources dynamically which allows assigning more memory than is physically available in the system and at the same time allows applications to request memory no earlier than needed - this optimization is not available in RT-context as memory allocation is not bounded (i.e. memory might not be available at the time it was requested sending the process back to sleep).**
- **Caching: The caching here referees to software caching (not hardware cache) by keeping pages in memory that were loaded from a slow mass storage media (hard-disk), access to frequently used data, libraries or applications can be optimized. This strategy requires large amounts of dynamic memory and also includes a processing overhead on a cache miss (flushing/freeing caches) thus RT-systems can't use this method (Note: a cache miss is the event of referencing a datum that is not available in cache and must be brought in from a secondary storage unig (i.e. hard-disk) - this is done by a page-fault in Linux as granularity of caches in Linux**

for application, libraries and user-data is generally on page boundaries).

- **Queueing: Instead of immediately honoring requests that are slow (i.e. write to a hard disk) requests are queued and then handled at once at some later time. This inherently does not allow deterministic behavior for the individual request making it unsuitable for RT-context.**

- **Reordering (out of order execution): for a user, interaction with the kernel are a series of resource requests , seemingly honored in the order we request them in. On a multitasking OS, many requests for the same resource may come in at the same time; the OS reorders them based on priorities (scheduler), may reorder their relative queue position to optimize head-positioning time of a hard-drive, or may reorder IP package based on the availability of a network link. All of these reordering strategies make the time until the request is honored non-deterministic.**

- **Fair scheduling: RT-systems obviously require a deterministic scheduling policy. Generally this means a strict priority based scheduling; this would let low priority processes 'starve' as long as there are high-priority processes runnable. In a GPOS we want a background task (i.e. delivery of an e-mail) not to be delayed indefinitely due to a compiler running. So the Linux kernel applies a scheduling strategy that raises the priority of a task if it had to wait, so sooner or later a task always ends up being the highest priority runnable task, which obviously is exactly the opposite of what an RT-system wants to allow.**

- **fast-path/slow-path strategy: Synchronization objects are used to protect concurrently accessed data objects. In most cases though this protection is only needed to catch the rare case of a conflict. In the majority of the cases there is no such conflict and thus the 'success path' for acquiring a synchronization object can be optimized; the failure path may though become substantially longer this way. As an example the Linux semaphore will decrement the counter before checking for a positive value, and only in case that, after decrementing the counter, a non zero value is present fix it again in the failure path. By doing this the fast path is reduced to decrement and compare. In a RT-system this would not be permissible as the worst-case delays are what matters and this worst-case**

would be given on failure to acquire the semaphore. Thus for a rt-system equal paths are anticipated even at the expense of reduced overall performance.

- Copy-on-Write: when a new process is created, the memory image of the parent process is not immediately copied, but Linux waits until the child process writes data to the memory image, thus making the memory different than the image of the parent. The copy is delayed until this inconsistency, as many processes never modify their memory image and copying would waste resources. This delay strategy leads to the stall of the child executing on the first write. RT-processes can't tolerate this; for a RT-process the memory image needs to be available unconditionally following process creation.

- Atomic operations: In a non-rt environment it can be tolerated to disallow context switches for a time by disabling interrupts. This allows kernel paths that need to perform complex operations to do these in a non-reentrant and thus simpler way, at the expense of the system delaying any possibly higher priority process during the execution of these operations. In a realtime environment these delays would directly be visible as scheduling/execution jitter, so such atomic, or uninterruptible, code paths must be kept very short.

0.5.10 User space applications

Any reasonable GPOS provides an isolation layer between privileged code and unprivileged code, this concept often refereed to as 'trusted code concept' was a design guideline for the entire UNIX operating system. Kernel code is trusted and user-space untrusted, that is there should be no implied guarantees for the behavior of user-space applications with respect to its effects on the OS Kernel, expected. So a user-space application never should be able to take down the system, but it should be allowed to fail in any way it wants without influencing any other user-space application via system services.

As desirable as this may obviously seem, it implies fairly heavy weight mechanisms to enforce the underlaying policies. This is not only a limitation when it comes to talking to hardware and mandating realtiem behavior it also is a performance issue especially relevant for embedded systems. User-space applications need to execute well-defined privileged functions - to do this they switch to kernel mode via the system-call interface, but passing data over

the kernel-space user-space boundary generally requires copying of data, which is expensive. There are ways to get around this copying by build zero-copy interfaces but one should be aware of the user-space 'inefficiency' being inherent to the concept of trusted and untrusted code and that violating this concept inflicts gravely on system security and possibly stability.

Aside from this a GPOS needs to provide user-space with a few general resources

- **hardware independent memory model**
- **reference resolution to allow dynamic loading of libraries**
- **abstraction of device related resources**

UNIX with its strong relating to the file concept can treat all of these issues via files `/dev/mem`, shared libraries and the dynamic linker loader (ld) and device abstraction `eth0`, `/dev/hda`. This means that a user-space application for a GPOS should ideally be totally hardware independent and the GNU/Linux system is well able to provide this.

0.5.11 User Interface

A critical issue for any GPOS that needs attention is the user interface, or human-machine-interface as automation people like to call it, although this is not a GPOS service, a GPOS must provide certain capabilities to allow application programmers to build such user-interfaces, with the goal to abstract the GPOS to a level where the user need not know anything about any detail of the GPOS - in fact ideally the user need not even know what GPOS is beneath the user-interface. Designing such a user-interface has consumed much effort in industry, with the consequence that one has achieved fairly good abstraction layers, but has forgotten to provide the necessary OS-interfaces that allow debugging, monitoring and clear post-mortem analysis. It should be the goal of designers of user-interface to split the task into two well defined and distinct parts:

- **User interface - providing access to the systems intended applications**
- **Administrative interface - providing access to the GPOS status, security state and resource management layer for monitoring.**

Especially for embedded and embedded realtime systems this split is an essential issue, GNU/Linux provides all necessary resources to build a powerful user-interface ?? but this split needs to be taken into consideration at project startup and during application design !

0.6 Guiding Standards

Some of the standards that should be considered as authoritative when looking into GPOS resources are:

- **POSIX.4 Realtime (TODO: find ref and full title)**
- **POSIX 1003.13 threads API**
- **susv2 (v3)**
- **LSB (TODO: ref)**
- **ELCPS (TODO: ref)**

Some of these are Linux specific, some are general UNIX, some OS independent, it is not the intention to give a full list here but only to list the ones that are relevant for the discussion here, not also that networking related standards were excluded (see ??)

Part I

Real Time Linux

Chapter 1

Introduction

In the first two parts of the report the underlaying technologies of the hard-realtime implementations of available Linux extensions are described. Basically there are two major RT implementations for Linux available.

- **Preemptive Kernel**
- **Dual Kernel concept**

As all available implementations of real time enhanced Linux are based on one of these two concepts an exhaustive description of the Preemptive Kernel in main stream Linux and RTLinux as the original implementation of the dual kernel concept are given prior to covering individual implementations.

The Hard-realtime implementations of available extensions all follow the dual kernel concept originally published by Victor Yodaiken and Michael Barabanov at New Mexico Tech ??. - Even though recent developments have signifficantly extended this concept (ADEOS) in the design, all currently available implementations follow the same methods (see the section on ADEOS for conceptual extensions). For this reason the RTLinux method is described conceptually first as the fundamentals apply to the other available implementations as well.

1.1 RTOS

There have been a number of proposed classifications for realtime systems

- **Hard vs. Soft Real Time**

- **Proprietary vs. Open**
- **Centralized vs. Distributed**

In this study we are concerned with hard as well as soft realtime, although the focus is on centralized realtime systems, extension via realtime enhanced networks is considered, in this limited sense distributed realtiem systems are covered. The issue of open vs. proprietary has shifted over time, in the 1990 one considered an OS open if it followed industry standards, in the context of this study open shall referee to open-source ?? systems vs. closed source proprietary systems (for insight on the problem of the term 'proprietary software' see ref[]).

The above classification, although commonly used, serves little practical purpose so it is only used with respect to document organization, a more relevant classification is to map areas of use to response time demands - although this is more useful for the practician it is also more fragile as for almost any field one can create exceptions to the given numbers here. Also note that the response time of a realtime system is not a sufficient classification for more on this problem see the section on Test concepts.

A further rough classification can be given based on the two leading design issues for GPOS vs. RTOS:

- **RTOS: maximize determinism**
- **GPOS: maximize average throughput**

These two fundamental goals impose some mutually exclusive restrictions on the design of the OS mechanisms.

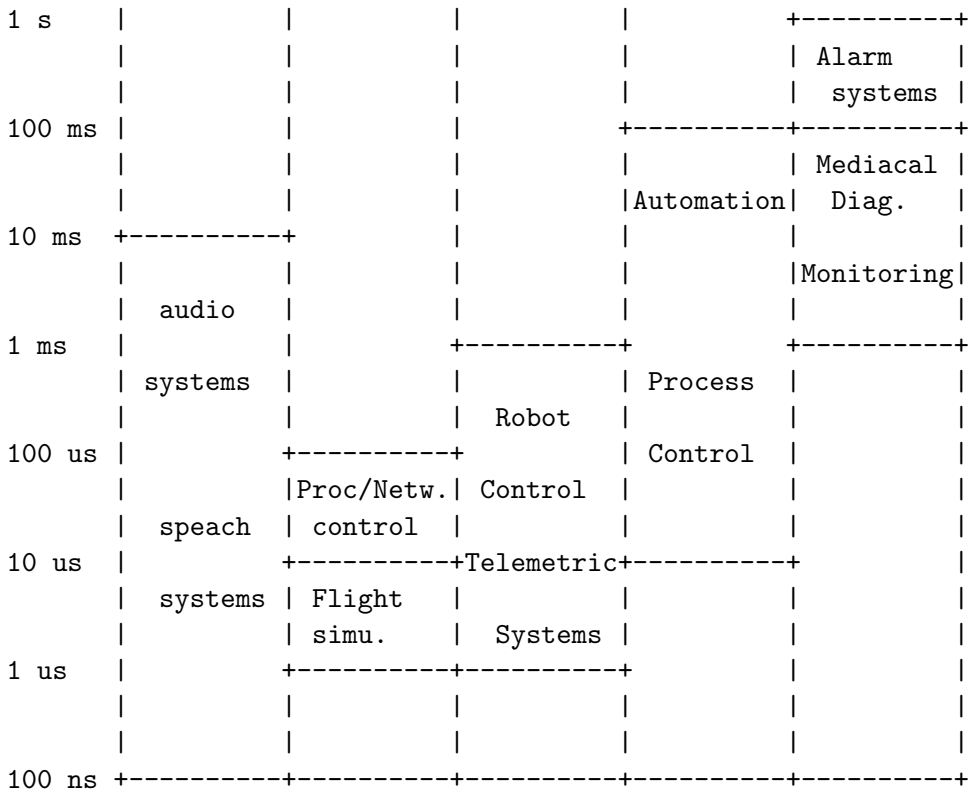

Figure 1.1: classifications for realtime systems

1.2 RTOS Design dilema

The fundamental problem of an RTOS is that users have conflicting demands with respect to system design. On the one hand, an RTOS should obviously be capable of realtime operations. On the other hand, users want access to the same rich feature sets found in general-purpose operating systems which run on desktop PCs and workstations. To resolve this dilemma, two general concepts, add GPOS featurs to an RTOS and modify a GPOS to be fully preemtible, have been used in the past.

1.2.1 Expand an RTOS

Design guidelines for an RTOS include the following: It needs to be compact, predictable and efficient; it should not need to manage an excessive number of resources; and it should not be dependent on any dynamically allocated resources. If one expands a small compact RTOS to incorporate the features of typical desktop sys-

tems, it is hard (if not impossible) to fulfill the demands of the core RTOS. Problems that arise from this approach include:

- The OS becomes very complex. This makes it difficult to ensure determinism, since all core capabilities must be fully preemptive.
- Drivers for hardware become very complex. Since blocking of a high priority rt-process on a non-rt process which has locked a specific hardware resource, refered to as priority inversion must not occur. Thus drivers must be able to handle situations in which servicing is intermediately delayed for posibly long periods.
- As complexity increases, dependencies become very complex. This makes systems hard to analyze and debug.
- Since the core system is an RTOS, the vast amount of free software that is available cannot (in most cases) be used unmodified without evaluating it with respect to RT-safety. It is even harder to use unmodified commercial software (where source code is not available), because it is almost impossible to determine interactions between the software and the RTOS.
- Many mechanisms for efficiency, like caching and queuing, become problematic. This prohibits usage of many typical optimization strategies for the non-realtime applications in the system.
- Maintenance costs of such a system are considerable for both developers and customers. Since every component of the system can influence the entire system's behavior, it is very hard to evaluate updates and modifications with respect to the realtime behavior.

1.2.2 Make a General Purpose OS Realtime Capable

The most seemingly natural alternative strategy would be to add RT capabilities to a general purpose OS, but this approach meets constraints similar to those noted above. Problems that arise with such an approach include:

- General purpose operating systems are event-driven, not time-triggered.

- **General Purpose OS's are not fully preemptive systems. Making them fully preemptive requires modifications to all hardware drivers and to all resource handling code.**
- **Lack of built-in high-resolution timing functions entail substantial system modification.**
- **Modifying applications to be preemptive is very costly and error-prone.**
- **The use of modified applications would also greatly increase maintenance costs.**
- **Optimization strategies used in general purpose OSes can contradict the RT requirements. For example, removing all caching and queuing from an OS would substantially degrade performance in areas where there are no realtime demands.**
- **Because such systems are very complex (and often not well-documented), it is extremely difficult to reliably achieve full preemption in such a system.**

General purpose operating systems are efficient with resources. Because they don't manage time as an explicit resource, trying to modify the system to do so violates many of its design goals, and causes components to be used in ways they were never designed for. This, one could speculate, is in principle a bad strategy to achieve hard-realtime performance, but is a suitable strategy for soft-realtime demands (see Part II).

1.2.3 GPOS vs. RTOS performance

The above could be read as suggesting that a GPOS is generally not usable for realtime - so why are they being used ?? Clearly GPOS have some advantages over RTOS:

- **easier to program and debug as no temporal debuging is generally required.**
- **lots of software packages available**
- **offer better average performance than RTOS**
- **easier to modify and maintain**

Generally one can say that a GPOS system will always outperform a Soft-Realtime system which will outperform a Hard-Realtime system with respect to average stem throughput and

even general responsiveness. One should thus always evaluate first if a GPOS can perform the requested service before considering any extensions that add soft or hard-realtime capabilities and thus mandates increased CPU resources.

1.3 Dual Kernel concept

To resolve these conflicting demands, a simple solution has been developed. The basic concept originally implemented in RTLinux (ref) 1997 is to split the OS entirely – into one part that runs as a general purpose OS with no hard realtime capabilities, and a second part that is designed around these realtime capabilities and which reduces all other features to a bare minimum. This approach allows the non-realtime side of the OS to provide all the goodies that Linux desktop users are used to, while the realtime side can be kept small, fast and deterministic.

The Three fundamental concepts of RTLinux operation covered by a U.S. Patnet:

- **It disables all hardware interrupts in the general purpose OS - Linux.**
- **It provides interrupts via interrupt emulation to the general purpose OS, and direct access to hardware interrupts to be handled in real time.**
- **It runs the general purpose OS, non-realtime Linux as the lowest priority task - the "idle task" of RTLinux.**

So the RTLinux dual kernel strategy 1.2 is basically a dual-kernel concept where one kernel - the RT-kernel - has full control of the hardware and a non-RT general purpose OS is run as the idle task of the RT-kernel.

1.3.1 RTLinux Patent

The design notion embodied in RTLinux is not restricted to Linux-based systems. The original concept was aimed at finding a conceptually ideal solution to the above stated dilemma, and is OS-independent. The idea is covered by U.S. Patent 5,995,745 (1999), by Victor Yodaiken, and was first implemented for Linux by Michael Barabanov in 1996.

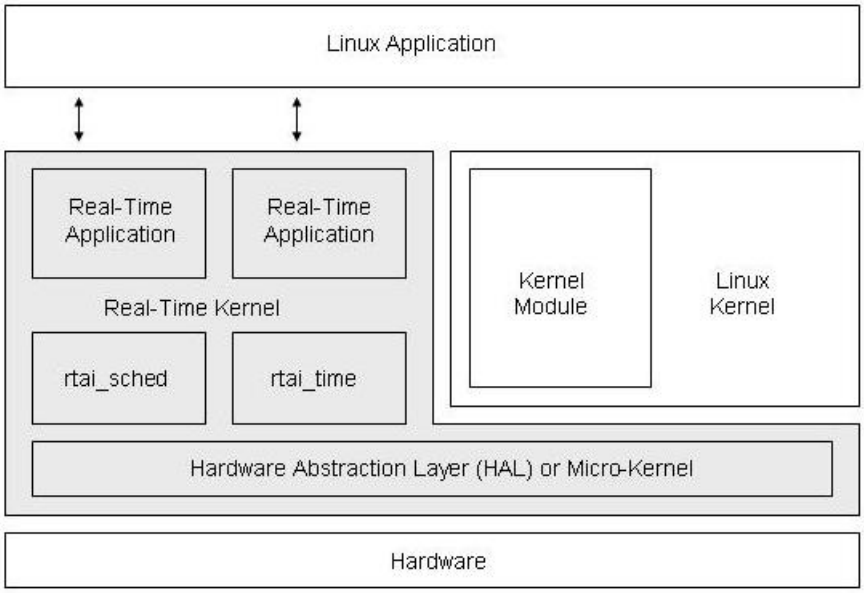

Figure 1.2: Dual Kernel Concept ([32])

What the Patent Covers

The patent covers the three essential components of the approach, as noted above:

- **Disable all interrupts in a general purpose OS**
- **Interrupt emulation**
- **Run the general purpose OS as the lowest priority task of the RTOS**

1.4 The RT-executive

The RTLinux executive, sometimes also called a micro or nano-kernel, provides an isolation layer between the hardware and the GPOS.

The core services of this executive are:

- **Interrupt Emulation**
- **Scheduling**

Interrupt Emulation

The main problem in adding hard real-time capabilities to the Linux operating system is that the disabling of interrupts is widely used in the kernel for synchronization purposes. The strategy of disabling interrupts in critical code sequences (as opposed to using synchronization mechanisms like semaphores or mutex), is quite efficient. It also makes code simpler, since it need not be designed to be reentrant. The monolithic Linux kernel has a flat memory structure and there are no internal boundaries in the kernel which protect memory of individual services or tasks.

The RT-executive runs in kernel address-space (above `0xC000000`) which has some implications that are note-worthy:

- **Real-Time tasks (threads) are executed inside kernel memory space, which prevents threads to be swapped-out to secondary memory**
- **The number of TLB misses is reduced due to a common address space (this does not though improve worst case performance).**
- **Threads are executed in processor supervisor mode (i.e. ring level 0 in i386 arch), and thus have full access to the underlying hardware.**
- **Since the RTOS and the application are linked together in a "single" execution space, there is no need for system calls to request privileged services , instead of using a software interrupt which produces higher overhead, the service request is reduced to a simple function call .**

there are some disadvantages to this approach as-well:

- **Lack of memory protection (how usable is memory protection in RT-context though, as aborting a task on meomry access violation is generally not an acceptable option see notes on memory protection in the section on user-space realtime)**
- **complexity of communication with user-space tasks**
- **limitations in the available resources (libraries wellies, secondary memory, some optimizations, etc.)**
- **Realtime applications require a high privilege level which obviously has security implications (ref to security section)**

Furthermore, the Linux kernel is not preemptive. That is, if a system call is in progress on behalf of a user space process, everything else must wait. This is good with respect to optimal resource usage and simplifies code development, but introduces substantial scheduling jitter and interrupt latency. As described above, modifying an existing multiuser/multitasking capable kernel to be fully preemptive would be difficult in a monolithic kernel like Linux. Considering the manner in which the Linux kernel is developed, with thousands of programmers coordinated (relatively loosely) via e-mail, such an effort would also certainly be very error-prone.

To maintain the structure of the Linux kernel while providing realtime capabilities one must provide an "interrupt interface" that give full control over interrupts, but at the same time appears to the rest of Linux like regular hardware interrupts. This interrupt interface is provided by interrupt emulation, one of the core concepts in RTLinux. Basically, interrupt emulation is achieved by replacing all occurrences of sti, cli and iret with emulation code. This introduces a software layer between the hardware interrupt controller and the Linux kernel. Note though that Linux *does* disable hardware interrupts in some very short sections even in RTLinux/RTAI for some hardware related management (MMU and trap-handling). These sections need to be reevaluated with new Linux kernel versions and sometimes patched to allow interrupt emulation to work properly. Currently none of the interrupt abstraction concepts really disable interrupts unconditionally in non-rt Linux rather the interrupts are evaluated on occurrence and ither propagated or delayed.

To guarantee hard realtime behavior without forcing substantial modifications on the non-realtime Linux kernel, all hardware interrupts must be handled by the realtime kernel (that is, the software layer between the hardware and the Linux kernel). Interrupts that are not destined for a realtime task must be passed on to the Linux kernel for proper handling when there is time to deal with them. In other words, RTLinux has full control over the hardware and non-realtime Linux sees soft_interrupts, not the "real" interrupts. This means that there is no need to recode drivers for Linux (provided there are no hard-coded instructions in the drivers that bypass the emulation). (See Section 1.5.)

Flow of Control on Interrupt **What happens when an interrupt occurs in RTLinux? The following pseudocode shows how RTLinux handles such an event - the actual code can befound in main/rtl_core.c**

(RTLinux/GPL V3.2-preX).

```
if(there is an RT-handler for the interrupt){
   call the RT-handler
}
if(there is a Linux-handler for the interrupt){
   call the Linux handler for this interrupt
}else{
   mark the interrupt as pending
}
```

This pseudocode represents the priority introduced by the emulation layer between hardware and the Linux kernel. If there is a realtime handler available, it is called. After this handler is processed, the Linux handler is called. This calling of the Linux handler is done indirectly–Linux runs as the idle task of the RTLinux kernel, so the Linux handler will be called as soon as there is time to do so, but a Linux interrupt handler cannot block RTLinux. That is, the interrupt handler for Linux is called from within Linux, not from RTLinux.

ADEOS extension of emulation **The Adaptive Domain Environment for Operating Systems, ADEOS, extents this concept to multiple domains and adds an API for managing interrupts in the context of an 'interrupt-pipeline'.ADEOS is a gereralized Hardware Abstraction Layer (HAL) designed to allow multiple OS, refered to as domains, to coexist indepnedantly of each other. The basic abstraction is the same just it is not RT-executive + Non-RT-executive + mark any other interrupts as pending, it becomes Highest priority RT-executive,2nd highest RT-executive.....Non-RT-executive. For the hard-realtime enabled Linux variants that utilize ADEOS to date (RTAI and LXRT) there is no difference between the above scheme showed for RTLinux and ADEOS, and to date the concept of multiple OS being managed by ADEOS has not been demonstrated (the multi-'OS' demo that does exist runs RTAI+Linux and within Linux it runs some OS-emulators, but *not* multiple OS/RTOS).**

ADEOS is a fast evolving technology, that has some quit interesting potentials targeted by the maintainers, like managing system calls through the ADEOS layer. It is to be expected that the development of ADEOS will speed up once it has been accepted as a replacement for the Real Time Hardware Abstraction Layer, RTHAL, patches to RTAI, currently both concepts are in use in the RTAI community (RTHAL for 2.4.X kernels ADEOS for 2.4.X and 2.6.X Kernels).

Limits of Interrupt Emulation Interrupt emulation has its limits. It must disturb the running realtime task to perform the emulation sequence, or the interrupt will be lost. The actual code is well optimized and has a platform-dependent runtime of less than 10 microseconds on x86 platforms. Scheduling jitter will increase if a system is put under very high interrupt load (e.g. if you ping-flood a system while running a critical realtime task). Thus, to test a system's worst-case scheduling jitter and interrupt response time, testing should be done under (at least) the same conditions as will be found during system operation. On the other hand, there is little sense in testing a system under unrealistic stress situations. Doing so will result in absolute worst-case values for the hardware, which if sufficient are safe values, but these might be far worse than ever reached during real operations.

Disabling Interrupts in Critical Sections To work around the jitter and latency introduced by the interrupt emulation, one can completely disable interrupts during critical sequences. This technique should not be used unnecessarily, since they can disturb systems (e.g. loss of network connection if the NIC is not serviced for too long, etc.), but provide a reliable means of securing extremely time-critical code sequences, or code that may not be interrupted without side-effects by hardware interrupts. Note though that this is a very limited method: it does not improve the timing precision of thread-startup and it will effectively prevent the scheduling of other realtime threads.

Note: it is a critical issue for a RTOS to provide tools to allow tracing of such sequences, if such tools are not available then debugging applications that utilize interrupt disabling become close to impossible. Especially for new evolving technologies like ADEOS it is a critical issue to watch - before such tools are available this technologies are to be considered experimental.

Basic structure of RT-processes

In most cases the realtime application will be split into a non-realtime part operating in regular user-space context or in non-rt Linux kernel context and a realtime executive, this split is true also for the 'user-space'realtime implementations as the rt-executives are always operating in a limited environment and thus certain services that can't be implemented (which a reasonable effort that is) in hard-realtime context are delegated to non-rt context, i.e user-interface, visualization , initialization and sometimes monitoring functions.

The realtime executives that generally communicate with the non-rt part of the application (see ??ommunication and ??ccess_resources below) can run in two distinct modes.

- **one-shot mode**
- **periodic mode**

Note that there is not pricipal difference between a one-shot mode and interrupt driven processing - just that with one-shot mode the interrupt comes from the hardware timer - thus one can see one-shot mode as the event driven and periodic mode as time driven processes.

one-shot processes In one-shot mode a timer is armed and the task is run when the timer expires, the basic structure of an application operating in one-shot mode is

```
while(1){
   do_something;
   expire_time+=interval;
   suspend_until(expiretime);
}
```

in RTLinux this would be coded as:

```
while(1){
   do_something;
   expire_time+=interval;
   clock_nanosleep(TIMER_ABSTIME,expire_time);
}
```

In RTAIs process model this would look like:

```
while(1){
   do_something;
   expire_time+=interval;
   rt_sleep_until(expire_time);
}
```

This again should show how similar the two implementations are - there is no fundamental difference between the two. RTLinux (both GP and Pro) are oriented towards the POSIX threads API, whereas RTAI has its own API which is not standardized but well readable.

One shot processes have to reprogram the timer at every interval if one implements a periodic process using a one-shot model, for non-constant intervals this is the simplest way to go.

periodic processes As many rt-processes are periodic processes all variants of hard realtime Linux support some way of coding explicit periodic processes, unfortunately POSIX does not have any POSIX compliant way of directly associating a period with a thread. For this reason non POSIX extensions are available in RTLinux and RTAI again uses a non-standard API all together so this does not bother RTAI.

A periodic process would be coded as:

```
rt_task_make_periodic(PERIOD);

while(1){
   do_something();
   rt_task_wait_period();
}
```

In RTLinux a periodic non-POSIX solution would look like:

```
pthread_make_periodic_np(pthread_self(),gethrtime(),PERIOD);

while(1){
   do_something();
   pethread_wait_np();
}
```

The POSIX compliant solution available in RTLinux/GPL is somewhat clumsy, and may explain why both RTLinux and RTAI decided not to bother with POSIX when it comes to periodic processes....

The model is to set up an interval timer, the only periodic object POSIX offers, to trigger a signal every time the interval expires, and have the signal handler wake up a one-time thread every time.

The timer handler just wakes up the rt-thread and relinquishes the CPU;

```
int timer_intr(int sig)
{
  pthread_kill(pthread_self(),RTL_SIGNAL_WAKEUP);
  pthread_yield();
}
```

The thread code needs to do some initialization of the signal handler and set the timer interval. The actual code is again the while(1) loop.

```
void *start_routine(void *arg)
{
  ...

  /* set up handler for the POSIX timer interrupt */
  sa.sa_handler=timer_intr;
  sa.sa_mask=0;
  sa.sa_flags=0;

  /* set up the interval timer */
  new_setting.it_interval.tv_sec=0; /* periodic */
  new_setting.it_interval.tv_nsec=1000000LL; /* period in ns -> 1kHz */

  new_setting.it_value.tv_sec=1;
  new_setting.it_value.tv_nsec=0;


  /* bind timer handler to signal */
  if ((err=sigaction(RTL_SIGUSR1,&sa,NULL))<0 ){
    rtl_printf("sigaction failed for RTL_SIGUSR1 (%d)\n",err);
  }

  err=timer_settime(timer,0,&new_setting,&old_setting);

  while(1){
    do_something();
    pthread_kill(pthread_self(),RTL_SIGNAL_SUSPEND);
    pthread_yield();
    pthread_testcancel();  /* honor any pending cancellation here */
  }
  ...
```

In the module initialization the signal is assigned and the timer aswell as the thread created.

```
int init_module(void) {
  ...
  /* timer should signal expiration via RTL_SIGUSR1  */
  signal.sigev_notify=SIGEV_SIGNAL;
  signal.sigev_signo=RTL_SIGUSR1;

  timer_create(CLOCK_REALTIME,&signal,&timer);
  pthread_create (&thread, NULL, start_routine,(void *) 0);
```

At first glance it may seem obvious that the non-POSIX way

is the better way , or simply that POSIX is not the right way.... we don' t see it that way because a closer analysis of the overall rt-thread system shows two essential conceptual advantages of the POSIX way:

- **all threads get the same structure**
- **time management and process is clearly separated**

What the first means is that it makes no difference if this model uses a POSIX interval timer a external hardware interrupt or a periodic internal hardware-clock as interrupt source - the concept stays unchanged.

The second allows to decouple time management (the timer setup and the timer signal handler) from the process executing periodically which simplifies the structure of the code and simplifies code analysis, the two problems can be cleanly slit.

Which concept is the best is to a large matter a question of personal taste, technically the three solutions are equivalent, with the exception that there is a small overhead for the timer+thread (POSIX compliant) solution but this overhead is marginal. Moving towards 'pure-POSIX' to us seems like the most suitable way of making rt-software well maintainable on a long term basis as the code is based on a well defined standard that is not expected to change without providing backwards compatibility, this may be considered the only rational argument for 'pure POSIX.

1.5 What happens to Linux

The RTLinux dual kernel concept is based on the POSIX realtime-extension that defines a single process running, in the context of this one process there are multiple threads of execution, one being the general purpose operating system.

In RTLinux, Linux is the GPOS to run below the RT-executive, as the GPOS may not ever prevent a RT-task to run when ready , the GPOS must be fully preemptible. This preempt-ability is achieved by running the GPOS as the lowest priority thread of the RT-executive (Priority -1, with RT priorities between 1 and 100000). This concept fews the entirety of the GPOS as one thread and allows no direct insight into the internal processes of Linux, so there is no direct way to talk to tasks in the GPOS (except for Kernel internal tasks).

RTLinux and Linux share the same address space, the kernel address-space also referred to as KERNEL_DS. This allows any RTLinux thread to directly access any kernel resources, which makes Linux-kernel RT-thread communication simple and effective, but RT-threads have no direct means of communicating with the address space of non-realtime Linux processes.

1.6 What happens to dynamic resources

Dynamic resources are a fundamental problem in real time systems, resources, especially CPU and memory allocation, can only be dynamic in a very limited way, that is the resource demand must be bounded to allow guarantees that they can be provided by the system, failure to provide resources demanded by an rt-thread would imply failure of the system as there is no real limit on the time it may take to free memory or to make the CPU available if resources were over committed.

This limitation is an inherent property of realtime systems. The consequence is that all resources must be allocated at task initialization and then locked (no swapping of memory to secondary memory etc.), simply speaking the task may not start if all resource demand are not satisfied. This demand for a hard-realtime system is one of the reasons that languages like C++ are not supported, or supported only with limitations, as C++ requires dynamic resources at runtime.

Specifically for memory allocation, there are strategies available to reduce the overall amount of resources in cases where the application can manage these internally and the maximum of memory ever needed is well known, in such cases a limited dynamic allocator can be applied (i.e. bget) and only the memory pool is allocated at task initialization.

Management of PCI resources can be done entirely with the PCI implementation of the GNU/Linux OS - the only limitation being that all PCI device configuration must be done from non-rt context (in the init_module section). PCI header read/writes during in RT context can be problematic and should be limited to the non-rt (Linux) context.

Allocation of the CPU to a specific task or reservation of a CPU for exclusive use by RT-threads is possible from RT-context.

1.7 Preemtive Kernel

Aside from the dual kernel strategy the second Linux related RT-concept is the preemptive kernel approach, as this approach has made it into the main stream kernel as of Linux 2.5.X and is firmly established with the 2.6.X version of the mainstream kernel. These soft-realtime variants are covered in part II exhaustively.

1.8 Overview of existing RT-extensions to Linux

For practical purposes the overview of existing solution was shifted to a separated part of this study - pleas referee to part V for an overview of hard real-time and soft-realtime variants.

The variants feature sets and basic data is presented in part V, variants listed there include:

- **Hard real-time micro-kernel extensions**
  - **RTAI/ADEOS**
  - **RTAI/RTHAL**
  - **RTLinux/GPL**
  - **RTLinux/Pro**
- **Hard real-time user-space extensions**
  - **PSC for RTLinux/GPL**
  - **LXRT for RTAI**
  - **PSDD for RTLinux/Pro**
- **Soft real-time kernel-modifications**
  - **Montavista Linux**
  - **Kurt**

For an introduction to the soft real-time variants referee to part II.

Basically the hard real-time variants can be split into two groups the RTAI and RTLinux based systems. The main differences between the two flavors of hard real-time enhanced Linux are:

- **RTAI has a very rich feature set, RTLinux is very conservative with respect to feature extensions**

- **RTLinux anticipates strict POSIX compliance, RTAI follows a self-defined API**
- **RTLinux is committed to backwards compatibility, RTAI will provide backwards compatibility but not at the expense of losing performance.**
- **RTAI develops patches independently resulting in somewhat hard-ware specific behavior and platform specific optimizations, RTLinux targets a unified, platform independent feature set.**
- **RTLinux is more conservative with respect to supporting the latest kernel releases, RTAI is known to move on to new kernels quickly and also support the development branches of Linux (i.e. 2.5.X, 2.6.X-testN).**
- **last but not least, RTAI, RTLinux/GPL and ADEOS are open-source projects, RTLinux/Pro is closed source and license based.**

We can't simply say which version is better, we do list recommendations along the way though to help make such a decision for a specific application. It should be noted though that the core technology below all hard real-time variants is identical (the interrupt abstraction layer and Linux as idle task). The main decision criteria in our view thus is the features required and the performance of the implementation aswell as the documentation and standards compliance.

Chapter 2

Kernel Space API

2.1 General

Describing the API is off course the respnsibility of the appropriated documents in the individual variants, generally this docuemtation is available for the API. Here we give a commented overview of the different APIs, sorted by:

- **threads**
- **signals**
- **interrupts**
- **timers**
- **IPC**
- **resource managment (see section ref Resources)**
- **synchronisation**

A number of functions listed here are marked with 'does nothing', these functoins should still be used in there appropriate place as the behavior may change in later releases. Typically the attribute destruction functions will simple be a `return 0;`, the funcions should be called any way as they are used in non-rt context (init_module) and thus the overhead of calling these empty functoins is ok.

2.1.1 Thread

POSIX threads, or pthreads, in RTLinux/GPL and RTLinux/Pro are based on the POSIX PSE 51 Minimum Realtime Profile. This profile introduces a single process per CPU and an arbitrary number of threads running in the common address-space of this one process. This model is basically folowed by all the dual kernel implementations of hard-realtime enable Linux, independant of the availability of a POSIX complient threads API or not. The model was originally introduced without this POSIX profile in mind ??araban_thesis, the common address space being mandated by the execution in Linux kernel context, which was chosen for a matter of efficiency and because all kernel-space implementations require access to Linux kernel functions (especially for interrupt managment). Even though RTAI stayed with the process model, again this was done for efficiency reasons, and to date RTLinux V1 API (on kernel 2.0.37) is still the fastest implementation (!), the model fits the PSE 51 profiles resource constraints well (not the API though).

The POSIX threads API is well designed and well documented, furthermore the requirements on the programmer are not as complex as she need not learn a completly new API, but can folow a well established API including non-rt variants ??inux threads being available for user-space.

Last but definitly not least, the available scientific publications that deal with behavior of POSIX threads semantics especially with respect to synchronisation are conciderable, so relying on POSIX threads is building on sound grounds.

This may give the impression that POSIX threads is the only resonable choice, and if pthreads would have been designed with realtime in mind we would see it this way, unfortunately pthreads were not designed with realtime in mind, and even more so POSIX singanls and timers, which are an important feature for building pure POSIX systems, are derived from the process model (see below). So it must be clear here, that pthreads are a good choice but there are limitations that need to be taken into account. We see these limitations as tolerable and the impact on performance as acceptable as it 'buys' complience to a well defined standard.

2.1.2 Timers

Every hardware platform has atleast one timer iterrupt source - IRQ 0 for Linux platforms that can be programmed to inter-

rupt the CPU at a specific time or with a cirtain period. POSIX was designed without a specific underlaying hardware concept, but rather treats timers as a general resource the the OS provides. This approach is confortable for the programmer but hard to implement efficient in the OS, for this reason currently only RTLinux/GPL implements POSIX timers.

Timers are used to allow for different types of timers, including implementation specific timer types. It also is usable for signal delivery as POSIX.4 only provides a sigalarm interface but no general interface for delivering one-shot events. Even more limiting is the lack of any notion of periodic execution in signals and thread-scheduling, POSIX can't directly provide periodic thread execution. The only object that has a periodic behavior associated with it in POSIX are timers (more specifically interval timers or itimers).

Naturally all flavors of realtime enhanced Linux must be able to offer preiodic execution AND one-shot execution. For those that don't provide POSIX timers different non-posix functions (pthread_make_periodic_np/
pthread_wait_np in RTLinux/Pro and rt_task_make_periodic/rt_task_wait
_period in RTAI) are provided within the respective API for periodic thread execution additionally RTAI offers non-posix timers (aswell as softreal-time timers implemented as Linux kernel tasklets). For one-shot execution, which are thread related implicid timers, variants of sleep (clock_nanosleep,usleep for the POSIX flavor in RTLinux and rt_sleep, rt_sleep_until,etc in RTAI) are available along side the posix timers in RTLinux/GPL tjat asp can be programmed as one-shot timers.

2.1.3 Interrupts

The basic mechanism of interrupt handling is described in the introductory sections on interrupt emulation, here we are more interested in the API for managment of interrupts. POSIX was not designed with respect to a specific hardware or with conciderations for hardware related issues, thus posix says littl about interrup managment facilities, as these are fairly cpu-specific. Never the less Linux has abstracted the interrupt capabilities of a large number of CPUs and manged to put a general interrupt managment API on top of this. All RTAI/RTHAL, RTLinux/GPL and RTLinux/Pro modify these functions but basically utilize the Linux functions (even if renamed or accessed via wrappers), ADEOS has a sligtly different approach as ADEOS anticipates a much

more elaborate interrupt handling (pipelining) concept, thus the ADEOS interrupt API is listed in more detail. All implementations willprovide a way of simply disabling interrupts and reenabling them (potentially losing interrupts, and a second method that allows saving the interrupt status register(s) so that on restore the previous flags/state is made available.

With the extension of real-time implementations to Multiporocessor systems interrupt managment becoms quite a lot more complex, not only must asynchronous notification betwen processort be taken into account but also the concept of processor affinity, the later refering to the 'locality' of a process with respect to the CPU it uses.

2.1.4 Signals

Signals and multitasking are closely related, all flavors of UNIX support signals as a means of asynchronous notification (in some cases signals are the only IPC mechanism). POSIX singnals were designed with a process model in mind which lead to the standard not being clear on the context in which signal handlers need to execute. The consequence of this is that POSIX signal semantics may vary between POSIX complient implementations.

A POSIX signal is equivalent to an interrupt or exception occurance, just that it is not handled directly by the CPU but managed via software layer in the RTOS.

Typical signal usags include:

- timer expiration signaled to a related thread
- I/O completion notification
- Inter-process notification (including waking and terminating processes)

Signals may have a lazy delivery behavior, queued or unqueued behavior, this is implementation specific and need to be taken into account when designing applications that utilize signals.

2.2 RTAI (both RTHAL and ADEOS)

RTAI supports it's own API derived from the RTLinux V1 API (non-posix process based API). The new features (message queues, mailboxes, etc) do not follow a coherent API and are incompatible

between them. The same feature is implemented in several ways with different systems calls.

RTAI maintains compatibility with the V1 RTLinux API but do provide some limited POSIX compatibility via a seperate module which provides partial POSIX 1003.1c. (PThreads) and 1003.1b (Pqueues). It should be noted that even those functions that folow a POSIX syntax may, in some cases, implement non-POSIX semantics, in thsi sense RTAI is to be concidered a non-POSIX RTOS.

2.2.1 Non-POSIX Kernel-space API

RTAI continues to folow the task (process-model) API introduced with the RTLinux V1 API (RTLinux V0.1-V1.3), the kernel-space API for task managment was strongly extended during development, but no attempt to squeez it into any of the posible standardization efforts was taken. For a full documentation of these functions refere to the RTAI manual [29].

task creation functions

Initialize a task structure creating a scheduable instance.

```
rt_task_init
```

scheduling functions

The scheduling functions are splitt in four groups, periodic task managment, one-shot task managment, task termination and task resume functions.

```
rt_task_make_periodic
  rt_task_make_periodic_relative_ns
  rt_task_set_resume_end_times
  rt_set_resume_time
  rt_set_period
  next_period

rt_sleep
  rt_busy_sleep
  rt_sleep_until

rt_task_wait_period
rt_task_yield
```

```
rt_task_suspend

rt_task_resume
rt_task_wakeup_sleeping

rt_get_task_state
```

time managment functions

note that RTAI uses ticks as the prime time quantity, not seconds divisions (nano-secons) like RTLinux/GPL and RTLinux/Pro - this clearly has the advanage that no 64bit arithmatik needs to be performed on time values, but has the disadvantage of the actual value not being easy to interprete aswell as being very hardware dependant.

```
rt_get_time
  rt_get_time_cpuid
  rt_get_time_ns
  rt_get_time_ns_cpuid
  rt_get_cpu_time_ns
```

hardware related task functions

On multiprocessor systems switching execution of a task from one cpu to another is quite expensive so the concept of cpu-affinity was introduced fairly early, RTAI implements cpu-affinity by setting a cpu mask.

```
rt_set_runable_on_cpu
```

Linux assumes that kernel tasks are not using the Floating Point Unit, FPU, if they do , and that includes rt-processes runing in kernel space, then this must be explicidly managed - note that RTAI also provides a way to inform the Linux kernel of fpu usage in non-rt linux-kernel context for services RTAI is utilizing.

```
rt_task_use_fpu
rt_linux_use_fpu
```

It also should be noted that if the fpu needs to be used in IRQ context then this must be managed by the programmer explicidly (brute force saving and restoring the fpu-registers !) or the computation must be delegated to a rt-process with fpu-usage marked.

2.2.2 Kernel-space POSIX threads API

The POSIX complient threads API is somewhat incomplete and it is implemented as wrapper functions to the task API, the POSIX threas support is a configuration option and this compatibility layer must be selected at compile time. It should also be noted that the threads API in RTAI does not target full POSIX complience, that is the enhancments/extensions available in the non-POSIX API are mapped to the threads API. De facto it is very hard at this point to write 'pure POSIX' in RTAI, and we see littl point in doing this as long as there is no clear comitment on the side of the developers to move towards a POSIX complient system. At time of writing this is not to be expected, so we do not recommend building RTAI based applications on the pthreads API. Notably some functions provide POSIX syntax but not POSIX semantics, which may be quite confusing, futher there is no clear path on the future and backward sompatibility of the POSIX API in RTAI. At the core of this lies the descision of the the RTAI developers to provide POSIX only as a 'addon' and not the prime API to target, this is also reflected in the fact that the official API document [29] does not cover the pthreads API at all.

POSIX threads functions

The pthread_ functions are provided as wrappers to the process model rt_task functions, the current implementation is questionable, with respect to standards compliance and with respect to performance. The pthreads API is also somewhat incomplete so it is hard to write 'pure POSIX' with the available functions in RTAI. Note though that the RTAI core developers don't anticipate providing a POSIX conform layer, in this sense the critizissm presented here is not legitimate from the standpoint of the RTAI API design, with our preference for POSIX we concider this critizissm legitimate as the provided wrapper API and some documents [49] suggest that RTAI can be utilized in a POSIX complient manner which clearly is not the case.

```
clock_gettime - wrapper to rt_get_time
nanosleep - always TIEMR_ABSTIME

pthread_create
pthread_exit
sched_yield
```

```
pthread_self

pthread_attr_init - default SCHED_OTHER !
pthread_attr_destroy - does nothing
pthread_attr_setdetachstate
pthread_attr_getdetachstate
pthread_attr_setschedparam
pthread_attr_getschedparam

pthread_attr_setschedpolicy
pthread_attr_getschedpolicy
thread_attr_setinheritsched
pthread_attr_getinheritsched

pthread_attr_setscope - usless as only PTHREAD_SCOPE_SYSTEM
                        is supported any way
pthread_attr_getscope
pthread_setschedparam
pthread_getschedparam

get_min_priority - should be sched_get_priority_min
get_max_priority - should be sched_get_priority_max
```

Note that the posix functions provide SCHED_OTHER (in fact the default being SCHED_OTHER) but the schduler does not handle SCHED_OTHER, rather anything unequal SCHED_FIFO is handled as SCHED_RR.

Due to the limited scope of this first part of our study, and in depth analysis of the POSIX complience or non-compliance aswell as performance issues was not posible, but ourfindings at this point indicate that the POSIX threads API wrapper in RTAI is quite non-POSIX in its semantics and in part in its syntax.

It is not recomended to build applications on the currently available POSIX threads API compatibility layer. We belive that it would cause more problems to utliize a POSIX like API that does not folow POSIX mandated behavior than to program in a obviously and intentionally non-POSIX environment - the non-POSIX task API provides a number of additional services and featurs which could justify the non-POSIX API, the current POSIX wrapers don't provide these additional RTAI featurs and don't provide a portable API - thus we see no indication to using the pthreads functions in RTAI at this point.

POSIX threads function extensions

Extensions within the threads API - note that POSIX permits these _np (non-portable) extensions, but naturally use of these eliminates the portability, or requires to write the appropriate wrapper functions for the system to which one wants to port the application. As of the current set of _np functions the hardware related ones are hard to get around (pthread_setfp_np) the rest should not be used if portability is anticipated.

```
pthread_wait_np
pthread_suspend_np
pthread_wakeup_np
pthread_delete_np
pthread_make_periodic_np
pthread_setfp_np
```

2.2.3 Signals

RTAI does not provide a direct signal API, comparable to the POSIX sigaction construction. RTAI allows registration of 'signal' function to be executed befor the task is executed and after the context switch occurs, thus this function is called in the context of the task it was registered with. The last lines of the RTAI scheduler are:

```
rt_switch_to(new_task);
if (rt_current->signal) {
   (*rt_current->signal)();
}
```

This signal function can be set by

```
rt_task_init - at task initialisation (last parameter)
rt_task_signal_handler - at runtime
```

to reset this signal function one calls `rt_task_signal_handler` with a NULL argument.

The singal function is called with interrupts disabled and can be used to manage any pending signals - the manuals do not give any instruction on how to access the pending signals though. It looks like this feature is more or less unused..

Note: The usage of this signaling function is undocumented, in fact we were not able to find a single instance where this was in use...

2.2.4 RTAI BITS - the real signals ?

RTAI does not directly provide signals, as noted above there are some signal related functions 'floating around' but the usage is unclear. The way RTAI implements signals (in the sense of asynchronous notification) is by the bits API. The bits API is an RTAI specific featur and not documented in the RTAI manuals - its documentation is in the form of a README file in the sources and in the example codes, an in depth study of RTAI bits was not posible in the framework of this first study phase (TODO: phase 2 check semantics of bits and there rt-characteristics).

RTAI bits module provides helper functions for managment of compound synchronizations objects (basically 32bits long that can be set in and/or relations). These flags or events can be waited on similar to semaphores.

Single tests operations provided:

```
ALL_SET - all bits set
ANY_SET - any bit set
ALL_CLR - no bit set
ANY_CLR - if any bit unset
```

Combined tests operating on two bit objects:

```
ALL_SET_AND_ANY_SET
ALL_SET_AND_ALL_CLR
ALL_SET_AND_ANY_CLR
ANY_SET_AND_ALL_CLR
ANY_SET_AND_ANY_CLR
ALL_CLR_AND_ANY_CLR
ALL_SET_OR_ANY_SET
ALL_SET_OR_ALL_CLR
ALL_SET_OR_ANY_CLR
ANY_SET_OR_ALL_CLR
ANY_SET_OR_ANY_CLR
ALL_CLR_OR_ANY_CLR
```

Bit operations provided:

```
SET_BITS     - set specified bits
CLR_BITS     - clear specified bits
```

```
SET_CLR_BITS - set to mask
NOP_BITS     - do nothing
```

The API for bits resembles something similar to the signals API, this section would probably better end up in the non-standard IPC, but as it is the only signal facility available in RTAI we perfered listing it here.

```
rt_bits_init - init a BITS object
rt_bits_delete - delete a BITS object
```

Like all synchronisation objects bits must be initialized and destroyed, reuse of bits without reinitialization - just like for any other synchronisation object - is deprecated.

```
rt_bits_reset - reset BITS and wake all tasks (signal)
rt_get_bits - return current value
rt_bits_wait_if - test but don't block
rt_bits_wait - test and wait on BITS (blocking)
rt_bits_wait_until - test and wait with absolute timeout
rt_bits_wait_timed - test and wait with relative timeout
```

This set resebles the test, signal and wait functinality. In standard synchronisation objects syntax the rt_get_bits and rt_bits_wait_if can be seen as a trylock, rt_bits_reset as the signal_ and the rt_wait_ functions (except wait_if) as the variations of blocking wait on the synchronisation object.

This form of implementing signals is very non-standard it is unclear how far such synchronisation objects can be formally analized in a given task-set. Currently there is no facility to trace bits induced dependancies and provide temporal analysis, we also were not able to find any theoretical works on issues like priority inversion regarding usage of bits. Thus we do not recomment using the bits facility within RTAI based projects, although we do see this as an interesting technology provided the lack of analytical tools and theoretical works is resolved.

2.2.5 Interrupts

Interrupt managment functions in RTAI are listed as service functions, they are not standard compoient in any way. Although this API summary applies to RTAI/RTHAL aswell as RTAI/ADEOS it should be noted here that ADEOS provides further extended interrupt control functions for domain managment - see the section on ADEOS at the end of this chapter.

RTHAL control functions:

```
rt_mount_rtai - initialize RTHAL layer
rt_umount_rtai - hand interrupt control back to Linux
```

Global functions:

```
rt_global_cli - disable interrupts on all CPUs
rt_global_sti - enable on all CPUs
rt_global_save_flags - save irq state and disable
rt_global_restore_flags - restore state and enable
```

IRQ managment functions for controling the Programable Interrupt Controler, PIC:

```
rt_request_global_irq - assign irq handler for
                                    non-local irqs
request_RTirq - for backwars compatibilty on X86
rt_free_global_irq - release irq
rt_startup_irq - initialize irq and endable
                 (calls linux kernel init function)
rt_shutdown_irq - shut down the irq
rt_enable_irq - enable PIC irq-request
rt_disable_irq - disable irq on PIC
rt_mask_and_ack_irq - maks irq and reenable PIC
rt_unmask_irq - unmask irq on PIC
rt_ack_irq - acknowledge without masking
```

Symetric MultiProcessing, SMP related interrupt managment: These are SMP specific aswell as smp-save versions of above functions (where needed), maintaining two versions is done for performance reasons as the SMP-save versions generally require more expensive synchronisation. Also note that all X86 based SMP systems provide an Advanced Programable Interrupt Controller, APIC, so interrupt managment functions need to be extended for these which include the Inter Processor Interrupts, IPI, which are used for asynchronous notification between CPUs.

```
rt_global_save_flags_and_cli - save irq state and disable
                                                (SMP version)
send_ipi_logical - send IPI to specified destination(s)
send_ipi_shorthand - wrapper for above (all,self, all but self)
rt_assign_irq_to_cpu - set irq-affinity
rt_reset_irq_to_sym_mode - reset irq-affinity
```

Functions to modify Linux (non-rt) interrupts:

```
rt_request_linux_irq - assign linux handler (can be a shared irq)
rt_free_linux_irq - remove handler
rt_pend_linux_irq - eumulate hardware irq to linux
```

Soft Interrupt functions:
Note that these are fairly X86 biased and must be emulated on other archs (i.e. PPC). Soft-interrupts are used by LXRT and other user-space services (FIFOs, Mailboxes).

```
rt_request_srq - request a soft-interrupt
rt_free_srq - releas a soft-interrupt
rt_pend_linux_srq - trigger a soft-interrupt in linux
rt_request_timer - install a hardware timer handler
rt_free_timer - reset timer handler
rt_request_apic_timer - setup hardware timer on APIC
rt_free_apic_timer - reset timer handler
```

2.2.6 Timers

RTAI has two sets of functions that it refers to as timers (somtimes quite confusing to people),

- **hardware timer related functions.**
- **timed execution of a specifid function - the non-POSIX version of POSIX timers, RTAI referes to these as timed tasklets or timer tasklets.**

timers - hardware timers

The RTAI system timer(s) are refered to as timers in the documentation, timers are provided in two different modes:

- **one-shot-mode -** `rt_set_oneshot_mode`
- **periodic mode -** `rt_set_periodic_mode`

These functions are called from within init_module to set there desired behavior. The purpose of this timer concept is to allow multiple threads to be managed by an optimal timer instance, this optimization to our understanding is usable for rate monotonic task-sets, and for common period task-sets with 'just-in-time'execution strategy. For a single task like shown below this probably makes littl sense. The timer settings are global to all rt-tasks running on the system.

```
#define TIMEBASE 10000000 - timer base frequency ‘timer granularity‘
#define DELAY (20*TIMEBASE) - the tasks delays are multiples of the
                                                              TIMEBASE

int init_module (void) {
   ...
   rt_task_init(&task, task_function, 0, 2000, PRIORITY, 0, 0);
   start_rt_timer(nano2count(TIMEBASE));
   rt_task_resume(&task);
   ...
}
```

The timer is started and associated with the task implicidly as the they are using a common time base, or timer granularity.

```
void cleanup_module (void) {
   ...
   stop_rt_timer();
   rt_task_delete(&task);
   ...
}
```

the task function looks no different than a simple task but here the DELAY value is a multiple of the timer period.

```
static void task(int t)
{
   while (1) {
      count++;
      rt_sleep(nano2count(DELAY));
   }
   rt_task_suspend(rt_whoami());
}
```

Note that this strategy requires that the entire set of rt-tasks be known at system design, adding in tasks can break this optimization.

The hardware timer managment functions in RTAI (refered to as Timer functions in the RTAI manual) are:

```
rt_set_oneshot_mode
rt_set_periodic_mode
start_rt_timer - 8254 timer on X86
stop_rt_timer - 8254 timer on X86
start_rt_apic_timer
stop_rt_apic_timer
```

Note that these are very X86 slanted functions.
Also it should be noted that the use of the POSIX threads wraper API sets periodic mode in its init_module (sounds like a bug to us).

The API for time value manipulation in RTAI is due to the fact that RTAI operates internally on ticks, that is the time-base of the hardware-clock and does not convert to nanoseconds by default, to simplify managment and to eliminate hardware dependancies RTAI provides conversion functions.

```
count2nano
nano2count
cout2nano_cpuid - SMP related variant
nono2count_cpuid - SMP related variant
rt_get_time - current time in tiks
rt_get_time_ns - converted to nano-seconds
rt_get_cpu_time - time from specific CPU (SMP)
rt_get_cpu_time_ns - convert to nano-seconds
```

The last set of functions RTAI manuals list under timer functions are the sleep() equivalent functions that suspend a task for a defined time.

```
next_period - get the next wakeup time
rt_busy_sleep - "spinn" until (on SMP)
rt_sleep - relative time
rt_sleep_until - absolute time
```

timed tasklets

RTAI timed tasklets are non-POSIX timers, they are implemented via RTAI tasklet vacility (in fact the rt_init_timer and rt_init_tasklet function are identical), timers in RTAI, also refered to as timed tasklets, are executed befor the scheduler proper is invoked.
The timer related API in RTAI:

```
rt_init_timer() - initialize the timer tasklet structure
rt_insert_timer() - insert the timer tasklet,
                    register it with the time managment task.
rt_set_timer_firing_time() - arm the timer
rt_remove_timer() - delete a timer
```

Note that for modifying settings related to timers the tasklet functions are used (i.e. the timer functions are just remaped #define rt_timer_use_fpu rt_tasklet_use_fpu):

```
static struct rt_tasklet_struct *timer;

int init_module (void) {
   ...
   prt = rt_init_timer();
   rt_insert_timer(timer, 1, expire_time, period, timer_function,
                                                          0, 1);
   rt_tasklet_use_fpu(timer, 1);
   ...
}
```

For completnes the timer tasklet functions are listed, the equivalend tasklet functions could be used just as well, this may change though in the future.

```
rt_insert_timer - insert timer in the timer tasklet lista
rt_set_timer_firing_time - arm the timer
rt_set_timer_period - set priod of the timer
rt_set_timer_handler - overwrite the timer handler passed
                                        at tasklet_init
rt_set_timer_data - set data filed in tasklet structure
rt_timer_use_fpu - safe/restore fpu context wehn invoked
rt_timer_delete - remove timer tasklet
rt_remove_timer - remove timer in rt-context\\
                (CLEANUP: check code on that)
```

As RTAIs API is intentionally symetric with respect to user-space RT and kernel-space RT. This symetry allows use of identical code in LXRT the user-space realtime extension. LXRT - LinuX RealTime is described in section LXRT. Since tasklet functions in linux kernel context (soft-rt/non-rt) have less synchronisation demands the rt_set_ above can be optimized, so there is a set of rt_fast_set_ fiunctions available to non rt-context tasks.

Use of these optimized variants is not recomended as it breaks the concep of symetric API and thus would ot allow easy migration from user-space rt (LXRT) to kernel-space rt (RTAI).

This breaking of the symetric API is critical as in many cases LXRT is a development tool for code that should later run in kernel context. For projects that originally plan to use LXRT at runtime sticking strickly to the symetric API allows moving to kernel-space if performance requires this later.

2.2.7 Backwars/Forwards Compatibility

Compatibility is not the prime concern of RTAI developers, this should not give the impression that one needs to rewrite applica-

tions for every subrelease, but changes to improve performance or add featurs that the community conciders usefull are done without compromises. In some cases this will break compatibility, but generally the rewrite effort is limited, though rerunning tests is mandatory. In general upgrading RTAI versions is no problem if one is upgrading to close releases (rtai-1.24.9 to rtai-1.24.10), upgrading over multiple versions one should not expect compatibility - especially it is insufficient to assume compatibility just because it compiles, syntactic equivalence does not suggest semantics being unchanged ! This drawback of RTAI comes at the advantage of more featurs and conceptually better target specific optimization. We concider it primarily a question of personal taste of develoeprs which they prefere.

As a recomendation for RTAI based projects we advise not to switch RTAI versions during a project due to limited backarwards/froward compatibility.

2.2.8 POSIX synchronisation

TODO: analyze the synchronisation objects and which are non-rt safe.

The POSIX synchronisation objects available in RTAI are listed here, atleast for some there POSIX complieance is not given, for others a more in depth study of the sources would be required which was not posible due to time constraints in ths first phase of the study.

- **Mutex:**
  The pthread_mutex implementation is very non-posix, and shows a number of inefficiencies in the implementation (long switch statements due to the introduced non-POSIX `mutexkind` **and debug types being unconditionally included.**

```
pthread_mutex_init - wrapper to semaphors
mutex_inherit_prio - set priority inheritance on a mutex
pthread_mutexattr_init - mutexkind = PTHREAD_MUTEX_FAST_NP
pthread_mutexattr_destroy - does nothing
pthread_mutexattr_setkind_np - non portable types
  PTHREAD_MUTEX_FAST_NP
  PTHREAD_MUTEX_RECURSIVE_NP
  PTHREAD_MUTEX_ERRORCHECK_NP
pthread_mutexattr_getkind_np
```

```
pthread_mutex_trylock
pthread_mutex_lock
pthread_mutex_unlock
```

- **Conditional Variables:**

```
pthread_cond_init
pthread_cond_destroy
pthread_condattr_init - does nothing
pthread_condattr_destroy - does nothing
pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast
```

as posix compatibility layer does not compile in 24.1.11 it is hard to say if these functions are realy available or not from code checks it looks like it - hofrat need to have it compiling befor uncommenting this subsection...

2.2.9 very non-POSIX sync extensions

These functions are made available to the programmer although they are clearly internal functions to the synchronisation object implementation.

```
priority_enqueue_task - queue task on mutex wait queue
cond_enqueue_task - queue task on condvar wait queue
dequeue_task - dequeue task from mutex wait queue
```

Use of such functionality is not recomended as the conceptual background for such low level manipulation is not given and code utilizing thes somewhat unexpected functions would be hard to understand and maintain, direct queue manipulation in a task-set using standard synchronisation objects seems very unnecesssary to say the least.

2.2.10 POSIX protocols supported

Priority inheritance is available via mutex_inherit_prio (non-POSIX function) within the POSIX wrapper API (TODO: check effects of mixed mode tasks and pthrads).

2.3 RTLinux/GPL

RTLinux is implemented as a POSIX 1003.13 "minimal realtime profile" (PSE 51) threads API. The internal design was driven by the POSIX requirements. There are some non-POSIX extensions by design and some are provided to allow optimization even if a POSIX complient solution is posible. This is especially visible with respect to periodic execution. As POSIX has no notion of periodic thread execution, this limitation can be overcome in a standards complient manner using POSIX timers and signals but this introduces a cirtain overhead. Also, typically hardware specific optimization can not be provided within the framework of the POSIX standard (i.e. cpu_affininty,conditional floating point register store/restore operations, etc).

2.3.1 Kernel-space threads API

RTLinux/GPL currently provides the folowing POSIX complient threads API

```
clock_gettime
clock_settime
clock_getres
time
usleep
nanosleep

sched_get_priority_max
sched_get_priority_min

pthread_self

pthread_attr_init
pthread_attr_getstacksize
pthread_attr_setstacksize
pthread_attr_setschedparam
pthread_attr_getschedparam
pthread_attr_setdetachstate
pthread_attr_getdetachstate
pthread_yield

pthread_setschedparam
pthread_getschedparam
pthread_create
```

```
pthread_exit
pthread_setcanceltype
pthread_setcancelstate
pthread_cancel
pthread_testcancel
pthread_join
pthread_kill

pthread_cleanup_pop
pthread_cleanup_push

sysconf
uname
```

2.3.2 POSIX signals

The POSIX signals were developed in the framwork of the OCERA project at the university of Valencia (DISCA), there implementation is strictly POSIX oriented and a elaborate compliance test is included. As the POSIX signals incure a cirtain scheduler overhad for processing, they are provided as a compile time configuration option.

The POSIX signals in RTLinux are implemented as a 32bit signal 'register', a signal delivery means that a signal is marked in this 32bit value. When the scheduler is invokd it will, after selecting a task, check for any pending, non-blocked signals and process them if necessary. POSIX signals in RTLinux have a lazy delivery behavior, that is they will not call the scheduler to deliver signals imediatly on there own, if this behavior is anticipated then its up to the programmer to invoke the scheduler after having sent a signal to a thread.

RTLinux signal handlers execute in the context of the thread that the signal is deliverd to (invocation after the context switch occurs).

```
pthread_kill
sigemptyset
sigfillset
sigaddset
sigdelset
sigismember
sigaction
sigprocmask
```

```
pthread_sigmask
sigsuspend
sigpending
```

Via the `sigdelset`, `sigaddset` functions the signal mask can be set, signals can be ignored, blocked or delayed. RT-threads can wait for signal occurance and thus implement a POSIX complient periodic thread behavior using `sigwait`. RTLinux uses signal numbers below 7 internally thus signal numbers should not be used but the appropriate macros RTL_SIGRTMIN (9) and RTL_SIGRTMAX (31) for assignment of application specific signal numbers.

The signal numbers used by RTLinux by default, these signal numbers should not be reassigned by any application.

```
RTL_SIGNAL_NULL 0
RTL_SIGNAL_WAKEUP 1
RTL_SIGNAL_CANCEL 2
RTL_SIGNAL_SUSPEND 3
RTL_SIGNAL_TIMER 5
RTL_SIGNAL_READY 6
```

For application specific signaling purposes RTL_SIGUSR1 and RTL_SIGUSR2 are provided aswell as signals between RTL_SIGRTMIN/RTL_SIGRTMAX.

```
RTL_SIGUSR1  (RTL_SIGNAL_READY+1)
RTL_SIGUSR2  (RTL_SIGUSR1+1)
RTL_SIGRTMIN (RTL_SIGUSR2+1)
RTL_SIGRTMAX RTL_MAX_SIGNAL
```

2.3.3 Interrupts

RTLinux/GPL provides direct interrupt managment funcitons that are intentionally only for rt-drivers, there should be no reason to use this API for thread sychronisation - for that purpose POSIX compatible spinlocks (pthread_spinlock) are provided. For notes on the dispatch process of interrupts see the introductory section on interrupt emulation.

Global interrupt hardware managment functions:
In thread code these should generaly be used in the form of `pthread_spinlocks`, for hardware drivers and some initialisation code these may be needed though.

```
rtl_no_interrupts - disable and save state
rtl_restore_interrupts - enable and restore
```

```
rtl_stop_interrupts - disable (dangorous)
rtl_allow_interrupts - enable
```

Interrupt managment functions - driver related for assigning handlers and managing specific interrupts.

```
rtl_request_irq(3) - assign handler
rtl_free_irq(3) - release handler
rtl_hard_disable_irq(3) - disable specific interrupts
rtl_hard_enable_irq(3) - enable specific interrupt
```

These rtlinux specific functions are described in the man pages of section 3 of the rtldoc package.

Soft Interrupt managment functions:
This allows emulating hardware interrupts to linux. Soft interrupts are not delivered imediatly but are delayed until the enxt hardware interrupt destined for Linux arives - on idle systems the worst case delay of a soft-interrupt thus reaches the time defined by the `HZ` variable in Linux (default HZ value on X86 is 100 -> 10 milli-seconds). The `HZ` variable is the frequency at which the timer interrupt (IRQ0) is triggert by a periodic mode timer, on an idle system this interrupts constitutes the de-facto response granularity.

```
rtl_get_soft_irq - request a soft-interrupt
rtl_free_soft_irq - free a soft-interrupt
rtl_global_pend_irq - mark an interrupt for Linux
```

2.3.4 POSIX timer

POSIX timers come in two flavors:

- one-shot timers
- periodic timers, refered to as intervall timers

The RTLinux POSIX timer implementation done by the OCERA team, support:

- Support for additional clocks - implementation specific timers
- Allow time resolution to the hardware limit (generally nanoseconds by now)
- more flexible signal delivers (POSIX.4 only provides a single SIGALARM signal).

Currently the CLOCK_REALTIME is the only clock mandated by POSIX.4, thus for portability reasons this is the prefered clock to use in timer code. In cases where this is not done it should be noted explicidly.

```
timer_create
timer_settime
timer_gettime
timer_getoverrun
timer_delete
```

POSIX timers incure a cirtain overhead in the scheduling code, thus they are a compile time option if not needed they should be disabled in the system to optimize performance (relevant probably only on relatively slow systems X86 below 133 MHz).

Also most of these functions are described, for example, in the Single UNIX Specification, Version 2 ??usv2
`http://www.opengroup.org/onlinepubs/7908799/index.html`

2.3.5 POSIX synchronisation

Not all of these synchrnoisaiton objects are non-rt safe , that is most of them CAN NOT be called safely from linux kernel context to sync with rt-threads. (TODO: analyze the synchronisation objects and which are non-rt safe). No detailed description is givien here as these are POSIX conplient implementations, thus one should refere to the appropriate documentation in the Single Unix Specification V2 [30] and the man-pages.

- Mutex

  ```
  pthread_mutexattr_getpshared(3)
  pthread_mutexattr_setpshared(3)
  pthread_mutexattr_init(3)
  pthread_mutexattr_destroy(3)
  pthread_mutexattr_settype(3)
  pthread_mutexattr_gettype(3)
  pthread_mutex_init(3)
  pthread_mutex_destroy(3)
  pthread_mutex_lock(3)
  pthread_mutex_trylock(3)
  pthread_mutex_unlock(3)

  pthread_mutexattr_setprotocol(3)
  pthread_mutexattr_getprotocol(3)
  pthread_mutexattr_setprioceiling(3)
  pthread_mutexattr_getprioceiling(3)
  pthread_mutex_setprioceiling(3)
  pthread_mutex_getprioceiling(3)
  ```

- Conditional Variables

```
pthread_condattr_getpshared(3)
pthread_condattr_setpshared(3)
pthread_condattr_init(3)
pthread_condattr_destroy(3) - does nothing
pthread_cond_init(3)
pthread_cond_destroy(3) - does nothing
pthread_cond_wait(3)
pthread_cond_timedwait(3)
pthread_cond_broadcast(3)
pthread_cond_signal(3)
```

- Semaphores
  Semaphors and signals are a messy thing - in RTLinux/GPL `sem_wait` can be interrumped by a signal (as mandated by the POSIX standard). This means that the `sem_wait` funciton must check if it exited du to a signal or by `sem_post`. If `sem_wait` is interrupted by a signal the signal handler is executed first and then the thread is makred ready..

```
sem_init(3)
sem_destroy(3)
sem_getvalue(3)
sem_wait(3)
sem_trywait(3)
sem_post(3)
sem_timedwait(3)
```

- POSIX spin locks
  This is the preferable way to manage interupt disabling/enabling in POSIX threads - calls to the direct `rtl_stop_interrupts, rtl_allow _interrupts, etc.` is deprecated for synchronisation purposes (see section on interrupts).

```
pthread_spin_init(3)
pthread_spin_destroy(3)
pthread_spin_lock(3)
pthread_spin_trylock(3)
pthread_spin_unlock(3)
```

- POSIX bariers:
  POSIX bariers are not yet integrated in the rtlinux cvs tree as of Spe 9

2003, they are expected to be merged into rtlinux-3.2 final release due by the end of 2003. Currently bariers are available as a patch to rtlinux-3.2-preX.

```
pthread_barrierattr_init
pthread_barrierattr_getpshared
pthread_barrierattr_setpshared
pthread_barrierattr_destroy
pthread_barrier_init
pthread_barrier_wait
pthread_barrier_destroy
```

Note the (3) (2) appended to the function names indicate that these are documented in the regular linux threads API man pages, these functions have no RTLinux specific syntax extensions.

2.3.6 POSIX protocols supported

```
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIO_INHERIT
```

POSIX options supported

```
_POSIX_TIMEOUTS
_POSIX_SPIN_LOCKS
_POSIX_SEMAPHORES
```

Non-portable POSIX extensions

Extensions to the kernel-space API of RTLinux/GPL that are none-POSIX, are marked by the _np extension to the function name. These extensions are primarily due to the limitations of the POSIX threads API

- POSIX threads API does not provide a standard complient way to execute threads periodically (the timer solution noted above executes the timer periodically which wakes the thread but the thread has no notion of periodic execution).
- no support for hardware related issues (FPU access, CPU assignment, etc).

```
pthread_attr_setcpu_np - assign the created thread to a parti-
                                                    cular CPU
pthread_attr_getcpu_np - get the CPU the thread is currently
```

```
                                                    executing on
pthread_wait_np - suspend the execution of the calling thread
                  until the next period (for periodic tasks).
pthread_delete_np - delete the thread in a rt-asfe way from
                non-rt context (providing a timeout mechanism).
pthread_attr_setfp_np - mark the created thread as using or not
                                                    using the FPU
pthread_setfp_np - mark the thread as using or not using the FPU
pthread_make_periodic_np - set timing parameters for periodic
                                                  threads execution
pthread_suspend_np - suspend the execution of the calling thread.
pthread_wakeup_np - wake up the thread
```

To build periodic threads without utilizing POSIX timers and signals the _np extensions to the API can be used , currently these are somewhat more effective due to the implementation details than the 'pure POSIX' solutions for periodic task execution, this should though only be relevant for low-end systems (X86 below 133MHz).

Never the less we recomend using the POSIX style API for portability and consistancy of semantics, even at the price of some performance loss.

2.3.7 Backwars/Forwards Compatibility

Note that 'pure-POSIX' is posible in RTLinux/GPL and the API development is focused towards improving POSIX compatibility and completness. RTLinux has provided backwards compatibility in the past all the way back to the V1 API (non-POSIX) but this compatibility is at the price of reduced performance. This reduction of performance is due to the backwards compatibility being provided via wrappers to the V3 API - so backwards compatibility is not using the original implementation. To utilize the full performance and stay compatible to future releases of RTLinux/GPL pure-POSIX is advocated.

We recomend not using the V1 API unless actually running on V1.X RTLinux systems. For projects utilizing the current API, POSIX should be the guiding coding standard.

2.4 RTLinux/Pro

The RTLinux/Pro API for the first release (Dev-Kit 1.0) is identical to the RTLinux/GPL V3.1 API, at which time the splitt between RTLinux/GPL and RTLinux/Pro occured.

2.4.1 Kernel-space threads API

The kernel space threads API for RTLinux/Pro is based on the POSIX API. As signals and timers are not implemented a 'pure POSIX' implementation of periodic threads is not posible. The RTLinux/Pro API preferes to offer periodic thread execution via the non-portable (_np) extensions to its API.

```
clock_gettime
clock_nanosleep
clock_settime
clock_getres
time
usleep
nanosleep

pthread_self
pthread_equal

sched_get_priority_max
sched_get_priority_min
sched_setscheduler - not documented (?)

pthread_attr_init
pthread_attr_destroy
pthread_attr_getdetachstate
pthread_attr_getschedparam
pthread_attr_getstackaddr
pthread_attr_getstacksize
pthread_attr_setdetachstate
pthread_attr_setschedparam
pthread_attr_setstackaddr
pthread_attr_setstacksize

pthread_create
pthread_join
pthread_detach
pthreed_cancel
pthread_testcancel
sched_yield
pthread_kill
pthread_exit

pthread_getcpuclockid
pthread_getspecific
pthread_setspecific
```

```
pthread_getschedparam
pthread_setschedparam
pthread_setcancelstate
pthread_setcanceltype

pthread_cleanup_pop
pthread_cleanup_push

pthread_getcpuclockid - POSIX time accounting

sysconf
uname
```

A further pthread function to provide access to non-rt Linux (the idle thread) is `pthread_linux`, it is a non-POSIX function that returns the thread ID of Linux. With the exception of the functions for periodic threads, the RTLinux/Pro API can be concidered complete, and POSIC complient. The long term direction of the API clearly is towards full POSIX complience. We recomend conforming to the POSIX threads programming model as stiktly as posible and not utilizing the _np funcitons if posible when programing for RTLinux/Pro as this will ensure a maximum forward compatibility as the POSIX model is the native implementation, non-POSIX functions will be draged on for compatibility but may be less efficient wrapper functions.

2.4.2 POSIX synchronisation

These functions are designed for synchronizing threads in rt-context, even though linux is the idle thread of the system not all synchronisation objects can be called in a safe way from within Linux context. (TODO: analyze the synchronisation objects and which are non-rt safe). For a detailed description of these POSIX conplient functions, refere to the appropriate documentation in the Single Unix Specification V2 and pthread man-pages provided with UNIX (Linux) (the numbers folowing the function name gives the man-apge section to search).

- Mutex Attribute functions, note that allthough the attribute related functions in some cases do nothing but return 0 there use is mandatory as these objects are opaque data types and the behavior of these functions may change in future releases.

  ```
  pthread_mutexattr_init(3)
  pthread_mutexattr_destroy(3)
  pthread_mutexattr_getpshared(3)
  pthread_mutexattr_setpshared(3)
  pthread_mutexattr_settype(3)
  ```

```
pthread_mutexattr_gettype(3)
```

- Mutex functions

```
pthread_mutex_init(3)
pthread_mutex_destroy(3)
pthread_mutex_lock(3)
pthread_mutex_trylock(3)
pthread_mutex_unlock(3)
```

- Priority inheritance and priority ceiling related mutex functions - note that usage of such protocols to "solve" synchronisation problems is deprecated and analysis of code making use of such priority changing protocols is hard, if not imposible.

```
pthread_mutexattr_setprotocol(3)
pthread_mutexattr_getprotocol(3)
pthread_mutexattr_setprioceiling(3)
pthread_mutexattr_getprioceiling(3)
pthread_mutex_setprioceiling(3)
pthread_mutex_getprioceiling(3)
```

- Condvar attribute functions

```
pthread_condattr_init(3)
pthread_condattr_destroy(3)
pthread_condattr_getpshared(3)
pthread_condattr_setpshared(3)
```

- Conditional Variables - pthrea_cond_signal is implemented via pthread_cond_broadcast.

```
pthread_cond_init(3)
pthread_cond_destroy(3)
pthread_cond_wait(3)
pthread_cond_timedwait(3)
pthread_cond_broadcast(3)
pthread_cond_signal(3)
```

- Semaphores

```
sem_init(3)
sem_destroy(3)
sem_getvalue(3)
sem_wait(3)
sem_trywait(3)
sem_post(3)
sem_timedwait(3)
```

- POSIX spin locks

```
pthread_spin_init(3)
pthread_spin_destroy(3)
pthread_spin_lock(3)
pthread_spin_trylock(3)
pthread_spin_unlock(3)
```

2.4.3 POSIX protocols supported

RTLinux/Pro provides regression tests suites that validate the protocol support
.

```
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIO_INHERIT
```

2.4.4 POSIX options supported

```
_POSIX_TIMEOUTS
_POSIX_SPIN_LOCKS
_POSIX_SEMAPHORES
```

2.4.5 Non-portable POSIX extensions

The _np extensions in RTLinux/Pro can be splitt into two categories, those that were added to overcome some limitations of the POSIX standard, and those that were added to provide actual extensions. As noted a few time all ready POSIX was designed without concideration for managing a specific hardware setup, and thus does not provide any means for low level configurations, so the extensions doen for this purpose in RTLinux/Pro again cover the issues of associating threads with a specific CPU and managment of the FPU.

```
pthread_attr_getreserve_np
pthread_attr_setreserve_np - dissallow the GPOS on a specific CPU
pthread_attr_getcpu_np
```

```
pthread_attr_setcpu_np - schedule a thread on a specific CPU
pthread_attr_getfp_np
pthread_attr_setfp_np - mark the thread as using the FPU
pthread_setfp_np - alternative way of marking a thread using the FPU
```

The second group of _np functions is based on POSIX not having any notion of periodicity associated with threads, as periodic threads are a comon requirement and RTLinux/Pro does not provide POSIX timers, an extension to the pthreads API is provided that allows creating and managing periodic threads `pthread_make_periodic_np, pthread_wait_np`. The remaining three thread managment functions are not needed and infact is not recomended by FSM-Labs (man pthread_delete_np). as there is POSIX standard functionality for `pthread_suspend_np and pthread_wakeup_np` we don't recomend using these extension. The timer functions listed below are for completion only, they are provided for backwards compatibility and are to be concidered obsolete

```
pthread_make_periodic_np - make a thread periodic
pthread_wait_np - suspend a periodic thread
pthread_delete_np
pthread_suspend_np
pthread_wakeup_np
clock_gethrtime - obsolete: get hard realtiem from a specific clock
gethrtime - obsolete: get hard realtime
```

As RTLinux/Pro targets a POSIX threads API we recomend using the non-POSIX extensions only if necessary. Further the rational for there usage should be documented as to allow replacement with POSIX conform constructs when ported (or when provided by later versions).

2.4.6 Signals

RTLinux/Pro has a minimum POSIX complient singal API for managin internal signals and also hardware interrupts (which are treated internally like signals). The signal processing is done at the system level, there is no facility to assign a user-provided signal handling routine, rather the behavior on signal receive are predefined 'default handlers'. The sigaction interface is only available for associating interrupts with handlers but there is no signaling facility like POSIX signals available in rt-context.

```
pthread_kill - deliver signal
pthread_cancel - send cancelation signal to a thread
```

Signals supportd by `pthread_kill` are 0, RTL_SIGNAL_SUSPEND, RTL_SIGNAL_WAKEUP, RTL_SIGNAL_CANCEL. Signal delivery is not imediate but a signal is basically delivered by marking it as pending in the threads

signal mask, at the next scheduler invocation (next cancelation point) it will be honored. A special case is RTL_SIGNAL_CANCEL, for wich signal handling routines can be pushed and poped as cleanup handlers to ensure proper resource dealocation on asynchronous cancelation requests (i.e. releasing synchronisation objects).

```
pthread_cleanup_push - push a function to be called on cancelation
pthread_cleanup_pop - pop it off the cleanup stack
```

the sigaction facility allows to install general handlers to be invoked by hardware interrupt delivery wich RTLinux/Pro treats as signals delivered to user-space (see section on PSC). See the man pages for the given POSIX conforming functions.

2.4.7 Interrupts

RTLinux/Pro provides interrupt managment funcitons intentionally only for rt-drivers and for system configuration at runtime, these should be used with care. For thread sychronisation POSIX compatible spinlocks (pthread_spinlock) are provided and adviced [48]. Note also that the spinnlocks are SMP safe and thus make applications scalable.

Global interrupt hardware managment functions:
In thread code these should generaly be used in the form of `pthread_spinlocks`, for hardware drivers and some initialisation code these may be needed though.

```
rtl_no_interrupts - disable and save state
rtl_restore_interrupts - enable and restore
rtl_stop_interrupts- disable (dangorous)
rtl_allow_interrupts - enable
```

Interrupt managment functions - driver related for assigning handlers and managing specific interrupts, note that this can also be done in a POSIX complient way by use of the high-leve sigaction interface.

```
rtl_request_irq - assign handler
rtl_free_irq - release handler
rtl_hard_disable_irq - disable specific interrupts
rtl_hard_enable_irq - enable specific interrupt
```

Soft Interrupt managment functions:
This allows emulating hardware interrupts to linux. Soft interrupts are not delivered imediatly but are delayed until the enxt hardware interrupt destined for Linux arives - on idle systems the worst case delay of a soft-interrupt thus reaches the time defined by the `HZ` variable in Linux (default on X86 and PPC is 100 -> 10 milli-seconds).

```
rtl_get_soft_irq - request a soft-interrupt
rtl_free_soft_irq - free a soft-interrupt
rtl_global_pend_irq - mark an interrupt for Linux
```

Interrupt service routine error handling functions:
Just like `setjmp()` and `longjmp()` that are useful for dealing with errors in interrupt context in low-level subroutines, the `rtl_` are the rt-safe versions.

```
rtl_setjmp - save stack content to safe location
rtl_longjmp - jump back to saved context
```

Currently only RTLinux/Pro provides such error managment functions suited for rt-interrupt context.

2.4.8 Timers

RTLinux/Pro does not provide timers, instead the periodic thread execution extensions `pthread_make_periodic_np()`, `pthread_wait_np()` must be used to provide periodically invoked functions. It is our understanding that FSMLabs does not intend to extend the RTLinux/Pro API to include timers, and closely related to these, full POSIX signals.

2.4.9 Backwars/Forwards Compatibility

RTLinux/Pro is aiming at a POSIX compatible API and based on this API compatibility with future releases can be expected, backwards compatibility may be dropped at some point (that is backwards compatibility to the V1 API). The API will though not be pure POSIX due to the inherent limitations of the POSIX threads API noted above also some recent extensions to RTLinux/Pro (i.e. one-way queues - see section "RTLinux/Pro one-way queues") are non-POSIX complient extensions, the core-API though is to be expected to stay POSIX-threads complient.

Note that RTLinux/Pro and RTLinux/GPL are only compatible in core-functionality, compatibility does not extend to singals, timers, message queues and bariers, which are not available in RTLinux/Pro at this point. Aside from message queues it is not to be expected that these featurs will be added in the future due to the performance issues with signals/timers that FSMLabs sees as being critical.

2.5 ADEOS

ADEOS: Adaptive Domain Environment for Operating Systems

The adeos Kernel space API is limited to:

- interrupt managment

- interdomain communication

as its intention is to provide a configurable interrupt abstraction and emulation layer to several OS-layers. For other services, like kernel-space realtime or user-space realtime it relies on available implementations like RTAI or Xenomine. This is potentially the streength of the ADEOS concpet, it could provide a means of combining a number of different resources like OS-emulators and simulation tools or debuggers runing beneth a RTOS !

This is a technology in a fairly early stage, naturally with some problems, but it is expected that this will change fairly quickly. If the RTAI community adopts ADEOS as its prim technology for interrupt abstractio/emulation as replacment for the RTHAL concept, then ADEOS can be expected to be well maintained and stable. Plans to move RTLinux/GPL to ADEOS are also in the queue of the RTLinux/GPL maintainer.

The functions described here are for building new domains, to utilize the ADEOS concept for RTAI the available interfaces can be used, currently RTAI under Linux is the only fully ported ADEOS domain (some experimental ports of OS emulators have been done though).

We recomend building new projects that are basedon the X86 architecture and RTAI on the ADEOS technology, and not on the RTHAL. At time of writing it should be epected that this technology may have some startup problems still, but the mailing lists and the developers are fairly active so bug-fixes (if any) are provided quickly.

Aside from pure use as RTAI interrupt emulation layer, ADEOS is of interrest for operatio of RTOS and OS emulation layers aswell as for combining existing unrelated OS technologies on a single platform. The API presented below if the ADEOS internal API for writing such ADEOS enabled domains.

2.5.1 Interrupts

These functions are for programing of ADEOS domain interfaces, that is for building a ADEOS domain, they are not actual application functions, in this sense these functions are inherently non-standard, but that is true for all OS internal functions.

Global domain managment function:

```
adeos_register_domain - register domain in interrupt pipeline
adeos_renice_domain - change priority ("SCHED_RR" if newprio==oldprio);
adeos_suspend_domain - notify adeos, domain donea
adeos_hook_dswitch - install domein switch handlerG
```

Global interrupt functions, these applie to all registered domains:

```
adeos_alloc_irq -  Allocate a virtual/soft pipelined interrupt
adeos_free_irq - unregister interrupt
```

```
adeos_trigger_irq - generate soft-interrupt
adeos_trigger_ipi - genreate inerprocessor soft-interrupt
adeos_propagate_irq - pass irq down the pipeline
adeos_critical_enter - globally protected code
adeos_critical_exit - exit protected code
```

Interrupt setup functions:

```
adeos_virtualize_irq - atach handler for current doman
adeos_control_irq - change irq mode
adeos_set_irq_affinity - assign irq to specific cpu
```

Domain specific interrupt managment operations:

```
adeos_stall_pipeline - disable interrupts
adeos_unstall_pipeline - enable interrupts
adeos_restore_pipeline - enable interrupts with flags restored
adeos_restore_pipeline_from - as above, for given stage
adeos_stall_pipeline_from - stop deliver at give stage
adeos_unstall_pipeline_from - enable deliver beond give stage
adeos_test_pipeline - query own stage
adeos_test_pipeline_from - query speified stage
```

Combined interrupt operations:

```
adeos_test_and_stall_pipeline
adeos_test_and_stall_pipeline_from
```

Global hardware timer funcions:

```
adeos_tune_timer
```

2.5.2 ADEOS interrupt processing characteristics

The pipeline

The fundamental ADEOS structure, one must keep in mind is the chain of client domains asking for interrupt control. A domain is a kernel-based software component (located in the root-domains kernel space) which can ask the ADEOS layer to be notified of:

- every incoming hardware interrupt,
- every system call issued by Linux applications,
- other system events triggered by the kernel code (see System events).

ADEOS ensures that events are dispatched in an orderly manner to the various client domains, so it is possible to provide interrupt determinism. This is achieved by assigning each domain a static priority (domains can change there priority with a renice call though). This priority value strictly defines the delivery order of events to the domains. All active domains are queued according to their respective priority, forming the "pipeline" abstraction used by ADEOS to make the events flow, from the most to the less prioritary domain. Incoming events (including IRQs) are pushed to the head of the pipeline (i.e. to the most prioritary domain) and progress down to its tail (i.e. to the less prioritary domain). Domains of identical priority are handled in a FIFO manner with respect to creation order (round-robin order can be achived by a domain calling adeos_renice_domain with the new priority equal to the old priority - thus moving its position in the pipeline amongst the equal priority domains).

In order to defer the interrupts dispatching so that each domain has its own interrupt log which gets eventually played in a timely manner, ADEOS implements the "Optimistic interrupt protection" scheme as described by Stodolsky, Chen, and Bershad (http://citeseer.nj.nec.com/stodolsky93fast.html) [56]. Note that this paper is one of the papers often refered to as prior work to Victor Yodaikens patent claims - As this paper describes one of the attributes claimed in the RTLinux patent (US Patent Nr. :5,995,745) we can't see why this would constitute prior work to the patented mechanism. It should further be noted that the soft-interrupt mask proposed by stodolsky is used for somewhat different purposes (namly to distinguish real-time from non-realtime and not to provide a fast path for the common case of uninterrupted protected areas) than in the interrupt emulation of the RTLinux patent although the mechanism is the same.

"Optimistic interrupt protection" is a optimisation of the fast-path - but not the worst case path in prinzipal. The underlaying assumption is that in most cases of critical sections, which are to be short, no hardware interrupt will disturb execution. This allows to optimize the system by not using the hardware interrupt masking capabilities on entry of the critical section but defere the masking of interrupts until a interrupt actually occurs, by introducing a software layer that checks if a given interrupt should be delivered or not.

From the RT domains point of view, even the long interrupt path of any lower priority domain (e.g. Linux) can be immediately preempted though, and this is what counts for us as far as preemption latency is concerned. The overhead here is the time consumed to switch domains whenever an interrupt needs to be delivered to the RT domain while the Linux domain was running

On the other hand, if you only look at the overhead brought to Linux seen as a standalone domain (i.e. no RT domain aside), the overhead does exist for the kernel, that's a fact. But it is not higher than the one incurred by the classic soft PIC trick when a hardware interrupt comes in and the soft PIC handler decides to dispatch it immediately because the kernel accepts interrupts. In such a case, there is no domain to switch, but you still pay the price of performing the

interrupt virtualisation chores. i.e.

HAL - interrupt emulation:

```
Primary IRQ trampoline ->
  Soft PIC handler decision / interrupt emulation ->
    Original Linux IRQ handler
```

ADEOS:

```
Primary IRQ trampoline ->
  adeos_handle_irq/adeos_sync_stage ->
    Original Linux IRQ handler
```

but you still have the choice to use hardware interrupt masking (adeos_hw_cli/sti et al.) to protect critical sections in the RT domain if you like.

This is not done for RTAI over ADEOS in its present implementation though because as of now, the performance penalty of applying strict pipelining rules to all domains including RTAI is acceptable. This way, we also keep the possibility of pipelining other domains more prioritary than RTAI like a debugger for instance.

Interrupt propagation

When RTAI runs over ADEOS, the ADEOS pipeline contains two stages, through which IRQs are flowing:

```
*IRQ* => [domain RTAI(prio=200)] ===> [domain Linux(prio=100)]
```

Therefore, the RTAI domain is first notified of any incoming IRQ, processes it, then marks (by calling adeos_propagate_irq(irq);) such interrupt to be passed to the Linux domain if needed.

When a domain has finished processing all the pending IRQs it has received, it calls a special ADEOS service which yields the CPU to the next domain (adeos_suspend_domain();) down the pipeline, so the latter can process in turn the pending events it has been notified of, and this cycle continues down to the less prioritary domain of the pipeline (via adeos_walk_pipeline) until the next domain that stalled the pipeline or end of the pipeline is reached.

The stage of the pipeline occupied by any given domain can be "stalled", which means that the next incoming hardware interrupts will not be delivered to the domain's handler(s), and will be prevented from flowing down to the less prioritary domain(s) in the same move. While a stage is stalled, interrupts accumulate in the domain logs, and eventually get played when the stage is unstalled.

ADEOS has two basic propagation modes for interrupts through the pipeline:

- In the implicit mode, any incoming interrupt is automatically marked as pending by ADEOS into each and every receiving domain's log accepting the interrupt source.

- In the explicit mode, an interrupt must be propagated "manually" if needed by the interrupt handler to the neighbour domain down the pipeline.

This setting is defined on a per-domain, per-interrupt basis. RTAI over ADEOS always uses the explicit mode for all interrupts. This means that each handler must call the explicit propagation service to pass an incoming interrupt down the pipeline. rt_pend_linux_irq() is a simple wrapper to this ADEOS service, allowing a RTAI handler to ask ADEOS to mark an interrupt as pending in Linux's own interrupt log. When no RTAI handler is defined for a given interrupt, the RTAI to ADEOS interface unconditionally propagates the interrupt down to Linux: this keeps the system working when no RTAI application traps such interrupt.

Enabling/Disabling interrupts

After having taken over the box, ADEOS handles the interrupt disabling requests for the entire kernel. This means disabling the interrupt source at the hardware PIC level, _and_ locking out any interrupt delivery from this source to the current domain at the pipeline level. Conversely, enabling interrupts means reactivating the interrupt source at the PIC level, and allowing further delivery from this source to the current domain. Therefore, a domain enabling an interrupt source _must be the same_ as the one which disabled it, because IRQ disabling/enabling operations are context-dependent.

In ADEOS releases up to r8, only the PIC level action was taken, but the per-domain lock has been additionally enforced since ADEOS r9, because it prevents really bad bugs from happening with some drivers which use constructs like this one:

- The driver (thinks it) masks all IRQs at processor level. The driver uses interrupt type X to operate.

```
linux_cli()
```

- An interrupt controlled by the driver occurs, but since Linux asked for an interrupt-free section, it won't be delivered yet.

```
<irqX occurs> => logged by ADEOS, not dispatched
```

- The driver specifically masks the interrupt source it controls at PIC level, then re-enables interrupts at processor level. The driver expects irqX _not_ to happen anymore, whilst releasing other interrupt sources.

```
mask_irq(X)
linux_sti()
```

Interrupt stack overflows

The calls to rt_disable_irq()/rt_enable_irq() you can read in the "shintr" example are aimed at preventing the stack of a running IRQ handler to be preempted recursively by interrupts piling up, which might lead to a stack overflow with the RTHAL.

The good news is that disabling the ethernet IRQ source to prevent stack overflows under interrupt flooding is useless in our case, because ADEOS leaves the interrupt source masked while running the domain handlers. The interrupt source remains masked until some domain in the pipeline decides to eventually unmask it (usually the Linux handler does this when it is done with processing the interrupt).

The single exception to this rule concerns the timer interrupt, which is kept unmasked during the propagation because of its criticality.

Interrupt sharing and determinism

However, keeping an interrupt source masked while the propagation takes place through the pipeline may jeopardize the real-time determinism for the RTAI handler.

Since ADEOS guarantees that no stack overflow can occur due to interrupts piling up, there is no need to disable the interrupt source in the RTAI handler. But you still want to re-enable it in the Linux handler, so that further occurences can be immediately dispatched to the RTAI handler as soon as they occur on behalf of the Linux domain.

So, shared interrupt would be written this way:

```
static void handler(int irq)
{
#ifndef CONFIG_RTAI_ADEOS
     rt_disable_irq(ETHIRQ);
#endif
rt_pend_linux_irq(ETHIRQ);
rt_printk(">>> # RTAIIRQ: %d %d %d\n", cnt, irq, ETHIRQ);
}
```

```
static void linux_post_handler(int irq, void *dev_id, struct pt_regs
*regs)
{
     rt_enable_irq(ETHIRQ);
rt_printk(">>> # LINUXIRQ: %d %d %d\n", cnt, irq, ETHIRQ);
}
```

(Note: This will work with both ADEOS release r8, r9).

This matter could look like rather cryptic sometimes, but it will be actually simpler in the long run, because ADEOS tends to "commoditize" interrupt handling and provides for consistent behaviour regardless of the kinds and number of client domains it controls [?].

2.5.3 Performance

Benchmarks to find out what the overhead of this strategy is in case the critical section is interrupted with respect to interrupt latency are not yet available but, initial measures have shown a propagation time of about 250ns (Celeron 1Ghz) from the hardware interrupt to the RTAI domain, including the time of the domain switch needed to preempt Linux. The current preemption latency tests with RTAI (24.1.12 and 3.0) show 20us worst-case in kernel mode and 55us in hard user-space RT mode (i.e. LXRT on a typical Celeron 800Mhz), which is quite close to the old RTHAL figures on the same hardware. Obviously, many parameters can alter these results and they highly depend on hardware factors.

Additionally, it was found that the average case figures are slightly higher with ADEOS compared with RTHAL, the worst-case though showed to be almost the same, with the bonus of termporal stability in ADEOS. (..Don't ask me to explain why, I just don't know! :o)..Philipp Gerum)

2.5.4 ADEOS IPC

facilities to syncronize betwen domains:

- mutex
- event catching
- interrupts (explicid pipeline handling within the domain).
- global variables (all domain are in kernel space)

mutexes

These mutexes are not application mutexes, but domain mutexes, that is for synchronisation between domains - thi API is for implementing domains (like RTAI) not for applications. Conceptually they are *only* for the protection of

critical sections, usage as general resource mutexes is problematic as ADEOS does protect against priority invesion if the mutex locking domain suspends itself without prior release of the mutex.

```
adeos_lock_mutex
adeos_unlock_mutex
```

The sleepq can link multiple domains. It's a LIFO handled list using the m_link field of the domain descriptor for linkage, the order is guaranteed by the pipeline behavior of ADEOS. As adeos_mutex_lock stalls the pipeline at the locking domains position the order of the sleepq is guranteed, thus wakeups are in order of domain priority. Until there is no more sleeper's in the queue, adeos_unlock_mutex() calls __adeos_signal_mutex().

If a mutex is held and adeos_suspend_domain is called, priority inversion will most likely happen. No domain should hold a mutex at domain suspension !

TODO: As of r9c4 if a domain renices it self just befor going to sleep on a mutex - no propagation currently here. (The implementation may well be broken in this case.) . When going to sleep on a mutex it should check if it's still the highest prioritary domain.

TODO: check mutex behavior on RR (via renice call) this currently looks like a problem with respect to propagation behavior.

TODO: cleanup handlers for mutexes have been proposed put are currently not yet integrated in the release of ADEOS.

Inter-domain Data exchange

Thread specific data managment functions 'ADEOS' IPC, this can be seen as a System V style SHM infrastructure.

```
adeos_alloc_ptdkey - register global key associated with a thread
adeos_free_ptdkey - free a threads key
adeos_set_ptd - set thread data
adeos_get_ptd - get pointer to a give key
```

Inter-domain soft interrupts

Virtual interrupts are handled in exactly the same way as hardware generated interrupts. Soft-interrupt generation is a very basic, one-way only, inter-domain communication system.

- adeos_alloc_irq - grab a free irq.
- adeos_virtualize_irq - attach a handler to a virtual interrupt number.
- adeos_trigger_irq - generate soft-interrupt passing it the virtual interrupt.

- adeos_schedule_irq - generate soft-interrupt passing it to the interrupt pipeline including the current domain - the irq delivery on scheduled irqs can be delayed until the next time the domain is switched in.

This mechanism allows two domain to signal unidirectional provided both perform a call to adeos_virtualize_irq.

2.5.5 System events

As listed in the events handled by the pipeline above, there are system events triggered by the kernel code to notify listeners from internal operations, i.e.

```
 /* IDT fault vectors */
#define ADEOS_NR_FAULTS         32
/* Pseudo-vectors used for kernel events */
#define ADEOS_FIRST_KEVENT      ADEOS_NR_FAULTS
#define ADEOS_SYSCALL_PROLOGUE  (ADEOS_FIRST_KEVENT)
#define ADEOS_SYSCALL_EPILOGUE  (ADEOS_FIRST_KEVENT + 1)
#define ADEOS_SCHEDULE_HEAD     (ADEOS_FIRST_KEVENT + 2)
#define ADEOS_SCHEDULE_TAIL     (ADEOS_FIRST_KEVENT + 3)
#define ADEOS_ENTER_PROCESS     (ADEOS_FIRST_KEVENT + 4)
#define ADEOS_EXIT_PROCESS      (ADEOS_FIRST_KEVENT + 5)
#define ADEOS_SIGNAL_PROCESS    (ADEOS_FIRST_KEVENT + 6)
#define ADEOS_RENICE_PROCESS    (ADEOS_FIRST_KEVENT + 7)
#define ADEOS_USER_EVENT        (ADEOS_FIRST_KEVENT + 8)
#define ADEOS_LAST_KEVENT       (ADEOS_USER_EVENT)

#define ADEOS_NR_EVENTS         (ADEOS_LAST_KEVENT + 1)
```

The structure for event communication is the adevinfo structure,

```
typedef struct adevinfo {

    unsigned domid;
    unsigned event;
    void *evdata;

    int propagate;              /* Private */

} adevinfo_t;
```

Events

The event monitors ar a simple array counting the number of listening domains on any particular event. This is just a cheap optimisation to save the I-cache

here and there, so that the event dispatcher is not called if no one cares to receive the current event.

Inter domain event managment operations:

```
adeos_catch_event - trigger event (soft-interrupt)
adeos_propagate_event - pass on event to next stage
```

2.5.6 Domain Debuging

No debugger, no tracer (yet), just oops reports and manual instrumentation.

However, ADEOS + kpreempt + lolat + LTT have been merged once in r9c2 which is available at:
`http://savannah.gnu.org/download/xenomai/fusion/adeos-combo-2.4.21-r9c2.patch`
This does not (yet) include SMP support though.

To debug internal ADEOS delays/jitter one needs to hand code timestamps into the kernel core taking IRQ-specific timestamps during the IRQ flow:

- stamp[irq][0] = upon each IRQ arrival in adeos_handle_irq()
- stamp[irq][1] = in adeos_walk_pipeline(), so that I could check that the acknowledge code was not bugous
- stamp[irq][2] = in adeos_sync_stage()
- stamp[irq][3] = in the client domain handler called from sync_stage

For the application layer limited debuing is available by a ADEOS safe printk (kernel/printk.c is patched for this purpose) basically by mapping the spinlock functions used to the adeos spinnlocks (note: this means that heavy printk will impact on temporal behavior).

2.5.7 ADEOS Domain Examples

TODO: (no multi-domain code available yet other than for domains running as linux processes - xenomine)

http://savannah.nongnu.org/cgi-bin/viewcvs/adeos/adeos/platforms/linux/examples/simple/adtest.c

Chapter 3

Accessing Kernel Resources

Realtime enhanced Linux has been focused on developing the RT-specific layer that operates below Linux - within this development communication between RT-threads and kernel as well as user-space have been quite limited, in part due to the inherent restrictions of a RTOS and in part due to the restrictions imposed by the API implementations. This section should also help develop the picture of realtime enhanced Linux variants being not only hard-realtime OS but offering a continuum of hard-realtime, soft-realtime, non-realtime tasks coexisting on the same hardware platform and thus providing a very flexible environment - naturally this flexibility comes at the price of mandating developers know-how level to be clearly beyond 'pure' application programming skills, but clearly OS-design basics are required to utilize the full potential of hard-realtime enhanced Linux.

RT-threads are operating in the same address-space (kernel address-space above 0xC0000000) as the Linux kernel itself, so it seems natural to investigate what capabilities within the Linux kernel could be made available to RT-threads as to enhance communication paths too and from user-space and non-rt kernel-space and to overcome some of the limitations due to non-available optimizations in RT-context. In this section a few of these, very non-portable, absolutely non-POSIX, paths are described. The main resources of interest to RT-threads being:

- Tasklets
- Kernel Threads
- Software interrupts
- Sharing Memory
- Accessing Non-RT facilities in kernel space
- 'misusing' System calls

For a fairly generic set of simple examples see the current RTLinux/GPL tree . It should be noted that these solutions are not only non-portable, as noted above, but may well be kernel version specific to a certain extent.

Although this study is not intending to be a tutorial for programmers we include a number of example in this section simply because there is no real summary of using kernel-facilities in conjunction with realtime enhanced Linux other than - this paper needs some additional comments/updates which is why it is included in part here.

In many realtime applications the main challenge for the programmer is to find the correct split between what is to be executed in rt-context and what can be executed in non-rt context. The predominant method of splitting tasks has been splitting tasks into hard-realtime rt-context and non-rt user context. In many cases a more fine gain split is desired, allowing hard-rt and different levels of non-rt execution. Furthermore many, especially embedded devices, show a large percentage of CPU usage in kernel-space (i.e. networking and backbone-devices) as opposed to desk-top systems that generally show Little processing in kernel-space and a clear dominance of user-space processing. For such "kernel-centric" devices operating non-rt tasks or functions in kernel-space and not switching to user-mode is a performance issue (expense of system calls and data-communication over the kernel/user boundary). The task of designing this split requires a basic understanding of the facilities available on the non-rt side of the system and how to communicate with these. In this section the focus is on accessing Linux kernel facilities from rt-threads. User-space tasks, and communication with these, are neglected as they are considered sufficiently documented in the standard RTLinux/RTAI documentation.

Generally, for all facilities that are available in the Linux kernel, the prime concern to a rt-system designer, is if these can be safely called from rt-context or not. A simple rule is that anything that only involves bit operations `set_bit`,`test_and_set_bit`, `clear_bit` should be absolutely safe. Any functions that require more complex synchronization need close analysis (or brute force testing) before they can be deployed. As far as oure analysis goes the kernel functions used in the examples in RTLinux/GPL `examples/kernel_resources` are safe from rt-context with RTLinux-3.2-pre3 and Linux-2.4.20.

3.1 kthreads

Kernel threads are a mechanism in the Linux kernel that allow threads of execution to run in the kernels memory space (kernel context) but be visible as regular tasks. This means they can receive signals and execute user-space calls with certain limitations/provisions. Here we are not so much interested with the details of kernel threads within the Linux kernel itself, but rather with how to interface rt-threads via kthreads to non-rt kernel-space and user-space.

3.1.1 simple example

This first example is not rt-specific, it only should give a framework of a kthread, and show the relation between kthread programming and regular user-space programming. Basically the difference is that to utilize a kernel_thread it is necessary to set up the execution environment which normally a user-space application need not bother with too much. This module declares a kernel function exec_cmd that is local to this module, a kernel thread is initiated passing this function as the routine to execute and a string via the arg pointer. The call to kernel_thread() initializes a task structure that is visible from user space (the pid of the process is printk'ed) and the thread routine (exec_cmd) is executed once. As we did not set up a specific context for this thread it runs in the inherited context of insmod and thus prints to the current console via the echo command, note that this thread is spawning processes within Linux which could interact with any user-space application, this thus resembles a 'prototype' for kernel-space user-space IPC. The thread routine is comparable to a regular user-space function that would call execve except for the privileges and the enabling of the kernels data section to store command arguments in `set_fs(KERNEL_DS)`. This also shows one clear danger of kernel threads - if they are not set up carefully with respect to privileges they can result in a serious security problem - for details on this give the kmod kernel_thread implementation in `kernel/kmod.c` a look. If an application should utilize kernel_threads then it is mandatory that the security policy for this application specifies a related profile to guide the security design of the kernel_threads - leaving kernel_threads security issues unattended will sooner or later (most likely 'sooner') lead to a security breach in the application !

```
#define __KERNEL_SYSCALLS__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/kmod.h>
#include <linux/errno.h>
#include <linux/unistd.h>
#include <linux/smp_lock.h>

#include <asm/uaccess.h>

int errno;

char cmd_path[256] = "/bin/echo";

static int
```

```
exec_cmd(void * kthread_arg)
{
   struct task_struct *curtask = current;
   /* we set up a minimum environment but note that we still inherit
    * the environment of who ever launched insmod of this module !
    * sounds dangerous ? - it is !
    */
   static char * envp[] = { "HOME=/root ",
      "TERM=linux ",
      "PATH=/bin",
      NULL };
   char *argv[] = {
      cmd_path,
      kthread_arg,
      NULL };
   int ret;

   /* Give the kthread all effective privileges.. */
   curtask->euid = curtask->fsuid = 0;
   curtask->egid = curtask->fsgid = 0;
   cap_set_full(curtask->cap_effective);

   /* Allow execve args to be in kernel space. */
   set_fs(KERNEL_DS);

   printk("calling execve for %s \n",cmd_path);
   ret = execve(cmd_path, argv, envp);

   /* if we ever get here - execve failed */
   printk(KERN_ERR "failed to exec %s, ret = %d\n", cmd_path,ret);
   return -1;
}

int
init_module(void)
{
   pid_t pid;
   char kthread_arg[]="Hello Kernel World !";

   pid = kernel_thread(exec_cmd, (void*) kthread_arg, 0);
   if (pid < 0) {
      printk(KERN_ERR "fork failed, errno %d\n", -pid);
      return pid;
   }
```

```
   printk("fork ok, pid %d\n",pid);
    return 0;
}

void
leanup_module(void)
{
   printk("module exit\n");
}
```

3.2 communicating with rt-threads

Even though the examples here use RTLinux, simply because they were release with RTLinux/GPL, the kernel related parts can be used unmodified in RTAI or RTLinux/Pro.

3.2.1 buddy thread concept

One of the many traditional communication mechanisms are signals. As rt-threads are operating in kernel memory space and are not available via the Linux kernel task-structure direct Unix-signals from user-space applications to rt-threads are not possible. Possibilities that have been shown in RTLinux examples are to install rt-handlers for fifos and trigger signals via these rt-fifos. In the following code an alternative concept that is intended to be expanded in the future is shown. This concept introduces a buddy-thread to each rt-thread that runs in kernel space as a kernel thread and thus is reachable directly from user-space via regular Unix-signals. The signal is still a two hoop job, a signal is sent to the kthread identified by the pid of the kernel process and passed on to the rt-thread via directly modifying the pending signals mask of the rt-thread structure or by using the RTLinux non-POSIX API `pthread_kill` and `pthread_delete_np`.

The folowing is a trivial RT-thread - note that it only suspens itselfe without having marked it as periodic or setting up a signal handler - the wake up is done via the folowing kernel thread.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_signal.h> /* RTL_SIGNAL_WAKEUP */
#include <linux/sched.h>   /* flush_signals() */
#include <linux/init.h>

static pid_t kthread_id=0;
static wait_queue_head_t wait;
```

```
static int rt_thread_state=1; /* got to initialize it to != 0 */
#define ACTIVE 1
#define TERMINATED 0
static int state=ACTIVE;

#define NAME_LEN 16
static pthread_t rt_thread;

static void *
rtthread_code(void *arg)
{
while (1) {
      rtl_printf("RT-Thread woke up\n");
      pthread_suspend_np (pthread_self());
   }
   return 0;
}
```

The code shown above is RTLinux specific, but it structurally is identical with what RTAI would do, exchanging the rtl_printf for rt_printk and pthread_suspend_np for rt_task_suspend would make it RTAI compatible.

The actual kernel thread code has a initializing preamble to set up the task related structures (kthread appear as tasks in Linux /proc filesystem and ps-tools so some setup is necessary). folowed by the actual runtime `while(1)` loop. In this loop the task suspends itselve with a sleep call and is woken by a UNIX-signal it received froma user-space process.

```
static int
kthread_code( void *data )
{
   struct task_struct *kthread=current;
   char thread_name[NAME_LEN];

   memset(thread_name,0,NAME_LEN);
   daemonize();

   /* wait for pthread_create of the finish so we are in sync */
   while (!rt_thread_state) {
      current->state = TASK_INTERRUPTIBLE;
      schedule_timeout(1);
   }

   /* take the address of the rt-thread as the unique name */
```

```
   sprintf(thread_name,"rtl_%lx",(unsigned long)&rt_thread);
   strcpy(kthread->comm, thread_name);

   /* make it low priority */
   kthread->nice=20;

   /* clear all pending signals */
   spin_lock_irq(&kthread->sigmask_lock);
   sigemptyset(&kthread->blocked);
   flush_signals(kthread);
   recalc_sigpending(kthread);
   spin_unlock_irq(&kthread->sigmask_lock);

   /* wait for signals to pass on in an endless loop */
   while(1){
       interruptible_sleep_on(&wait);
      /* if we got a SIGKILL terminate the rt-thread and
       * exit the loop
       */
      if(sigtestsetmask(&kthread->pending.signal,
         sigmask(SIGKILL))  ){
         pthread_delete_np(rt_thread);
         break;
      }
      /* else send a RTL_SIGNAL_WAKEUP to the rt-thread
       * and sleep on
       */
      else{
         pthread_kill(rt_thread,RTL_SIGNAL_WAKEUP);
         spin_lock_irq(&kthread->sigmask_lock);
         sigemptyset(&kthread->blocked);
         flush_signals(kthread);
         recalc_sigpending(kthread);
         spin_unlock_irq(&kthread->sigmask_lock);
      }
   }
   /* so cleanup module knows when to safely exit */
   state=TERMINATED;
   return(0);
}
```

This kernel thread is basically not RTLinux specific in any way except for the pthread_kill call to signal a wakeup to the rt_thread,

```
int
```

```
init_module(void)
{
   struct sched_param p;
   init_waitqueue_head(&wait);
   kthread_id=kernel_thread(kthread_code,
      NULL,
       CLONE_FS|CLONE_FILES|CLONE_SIGHAND );
   printk("rt_sig_thread launched (pid %d)\n",
      kthread_id);
```

The above part of init_module is not RTLinux specific aside from the declaration of struct sched_param p;, the rest of init_module is RTLinux specific as RTLinux is using a POSIX threads API and not the RTAI process API - 'translating'this from RTLinux to RTAI is trivial though and introduces no new concepts.

This again should show how similar these to implementations are with respect to there basic structure.

```
   rt_thread_state = pthread_create (&rt_thread,
      NULL,
      rtthread_code,
      0);

   /* set up thread priority */
   p . sched_priority = 1;
   pthread_setschedparam (rt_thread SCHED_FIFO, &p);
   return 0;
}

void
cleanup_module(void)
{
   int ret;

   /* delete the rt-thread */
   pthread_delete_np (rt_thread);

   /* send a term signal to the kthread */
   ret = kill_proc(kthread_id, SIGKILL, 1);
   if (!ret) {
      int count = 10 * HZ;
      /* wait for the kthread to exit before terminating */
      while (state && --count) {
         current->state = TASK_INTERRUPTIBLE;
```

```
            schedule_timeout(1);
        }
    }
    printk("rt_sig_thread exit\n");
}
```

Note that LXRT original implementation used a buddy thread concept as well, but this is not related to the examples presented here as the goal here is to access unmodified kernel resources. It should be noted though that anything shown here for RTLinux could be done in RTAI using LXRT functionality as well as accessing direct kernel resources in a comparable way. The approach using LXRT has clear advantages for RTAI based systems, not only because it provides more functionality than this direct access can provide but also because it provides a symmetric API simplifying programming. in this sense the concepts presented here can be seen to be somewhat RTLinux slanted.

3.3 tasklets

Tasklets are the replacement of the bottom half concept that was in use up to kernel 2.2.X (in 2.4.X BH are still supported - but are implemented via tasklets). The main properties of tasklets:

- tasklets can be scheduled with different priorities in Linux
- tasklets don't need to be reentrant
- the same tasklet will never run in parallel on SMP
- scheduling a tasklet multiple times before it actually runs does not cause it to run multiple times.
- different tasklets may run on different CPU's at the same time.
- tasklets run in interrupt context - thus with all limitations of an interrupt handler.

These properties make it fairly simple to write tasklets. The concept behind them is the same as with the former BH handlers, keep the interrupt or rt-thread small and put all processing steps that may be delayed into a tasklet. This has the obvious advantage of keeping the operation times with disabled/mask interrupts low - long ISRs are always a potential source of high jitter, utilizing DSR mechanisms can reduce this clearly.

Important for realtime enhanced Linux is that tasklets are run at every context switch to Linux, they are not delayed until the next hardware interrupt. Tasklets will run before any user-space application will get a chance to run, thus they are a high priority non-rt task that can be easily scheduled from within a rt-thread by calling schedule_tasklet() or schedule_hi_tasklet, whereby the later has higher priority than the first.

3.3.1 simple tasklet example

This first example is basically only a slightly modified version of `examples/hello/hello.c` the main change is the introduction of the tasklet code itself and the scheduling of the tasklet. Note that tasklets can be scheduled from rt-context and from Linux kernel context without any conflict as the scheduling is performed by bit-operations which are atomic.

The rt-thread "collects" data, in the example the `arg` to the rt-thread is used as datum, and sprintf's it to the tasklets data object `tasklet_data`. The `tasklet_data` is a simple example of a shared object between RT-context and Linux tasklets. The tasklet then is scheduled, thus marking it for execution as soon as the system switches back to Linux (non-RT mode). This setup can be used for maintenance purposes and, in a limited way, to implement dynamic resources. The rational behind this form of delegation is:

- Tasklets are executed imediatly after pending interrupts - so they are a fast path
- Tasklets may sleep
- Tasklets have full access to kernel resources
- Tasklets are light weight as they don't have a specific context requireing a context switch (just like ISRs in Linux)

Note though that tasklets scheduled multiple time befor they actually have a chance to run, are executed once only. This can easally happen in rt-context as the tasklet will not be executed until the system switches back to Linux context !

A tasklet is declared with the DECLARE_TASKLET() macro and scheduled with schedule_tasklet or schedule_hi_tasklet. The tasklet related macros are found in `linux/interrupts.h`.

```
#include <rtl.h>
#include <time.h>
#include <linux/interrupt.h> /* for the tasklet macros/functions */
#include <pthread.h>

int myint_for_something=1;
pthread_t thread;

void tasklet_function(unsigned long);

char tasklet_data[64];

DECLARE_TASKLET
```

```
(test_tasklet,tasklet_function, (unsigned long) &tasklet_data);

void *
start_routine(void *arg)
{
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

   pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

   while (1) {
      pthread_wait_np ();
      rtl_printf("RT-Thread; my arg is %x\n", (unsigned) arg);
      sprintf(tasklet_data,"%s \"%x\"",
         "Linux tasklet received RT-Thread arg",
         (unsigned) arg);
      tasklet_hi_schedule(&test_tasklet);
   }
   return 0;
}

void
tasklet_function(unsigned long data)
{
   struct timeval now;
   do_gettimeofday(&now);
   printk("%s at %ld,%ld\n",(char *) data,now.tv_sec,now.tv_usec);
}

int init_module(void) {
   sprintf(tasklet_data,"%s\n",
      "Linux tasklet called in init_module");
   tasklet_schedule(&test_tasklet);

   return pthread_create (&thread, NULL, start_routine, 0);
}

void
cleanup_module(void)
{
   pthread_delete_np (thread);
}
```

This simple example aside from showing the basics of implementing a tasklet also allows to see the delay times between rt-threads and tasklets, if run with only one short rt-thread as in this example coupling is naturally very good - to see the really coupling one could run this module together with the actual target application to get a fairly close picture of the delays introduced.

3.3.2 scheduling tasklets from rt-context

From `linux/interrupt.h`:

```
/* PLEASE, avoid to allocate new softirqs, if you need
   not _really_ high frequency threaded job scheduling.
   For almost all the purposes tasklets are more than enough.
   F.e. all serial device BHs et al. should be converted to
   tasklets, not to softirqs.
 */
```

The tasklet priority of a tasklet scheduled with `schedule_hi_tasklet` is above the network subsystem, so if you over due it you actually can cripple your network performance..., `schedule_tasklet` has a priority just below the network subsystem so a network overload can delay your tasklet substantially.

With the kernel functions `tasklet_disable` and `tasklet_enable` the execution of a tasklet can be suspended. If a tasklet was scheduled and is disabled before it was executed it will be executed when `tasklet_enabled` is called. For the full set of kernel functions available for tasklets check `linux/interrupt.h`, note though that you must check if these are safe to be called from rt-context, for this paper checks were done against Linux 2.4.4.

To ensure synchronization of tasklet scheduling when disabling tasklets within rt-context with `tasklet_disable` one must install a cleanup handler to reenable the tasklet on termination of the thread so that a scheduled tasklet can be executed and the tasklet structure can be removed on module exit.

```
...
void
tasklet_cleanup(void *arg)
{
   tasklet_enable(&test_tasklet);
   rtl_printf("cleanup handler called\n");
}

void *
start_routine(void *arg)
{
   ...
```

```
   pthread_cleanup_push(tasklet_cleanup,0);

   while (1) {
      pthread_wait_np ();
      ...
      if(i==20){
         tasklet_disable(&test_tasklet);
         rtl_printf("killed tasklet\n");
      }
      tasklet_hi_schedule(&test_tasklet);
      i++;
   }
   pthread_cleanup_pop(0);
   return 0;
}
```

This somewhat artificial code shows the basic setup - a cleanup handler to reenable the tasklet is installed and within the main loop of the rt-thread `tasklet_disable` is called to disable the test_tasklet, the cleanup handler is executed on termination of the `while(1)` loop and reenables tasklets.

3.3.3 naive rt-allocator

As a second somewhat more interesting example of using a tasklet from rt-context a naive rt_allocator framework is presented. The tasklet is called from a rt-function that suspends the running thread `rtl_malloc(size)`, this allocator will call a tasklet to do the actual memory allocation and then signal `RTL_SIGNAL_WAKEUP` back to the rt-thread when the allocator thread is done. The allocation thus is non-realtime and the realtime thread needs to check if memory actually was allocated successfully or not. Note that the call to kmalloc in the tasklet uses the flags GFP_ATOMIC which is necessary, if GFP_KERNEL were used the tasklet could sleep and thus the system would hang.

This allocator has a automatic initialized array of pointers to char set and will allocate a requested size of memory assigned to these pointers. These are globally available so the tasklet can signal a wakeup to the rt-thread by setting the appropriate bit in the threads pending signal mask. instead of setting the bit directly one could also call pthread_kill(rt_thread,RTL_SIGNAL _WAKEUP), if modules are split between kernel and RTL context it sometimes is a problem to include RTL API-calls that require RTL-header files so in those cases directly accessing the signal pending mask solves the problem.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
```

```
pthread_t rt_thread;


#include <linux/interrupt.h> /* for the tasklet macros/functions */
#include <linux/slab.h>      /* kmalloc */

void allocator_function(unsigned long arg);
#define BUFFERS 128
static char *iptr[BUFFERS]; /* static array of pointers for the buffers */
static int iptr_idx;
DECLARE_TASKLET(allocator_tasklet,allocator_function,0);


void
allocator_function(unsigned long arg)
{
   struct timeval now;
   do_gettimeofday(&now);
   printk("tasklet: allocating %ld at %ld,%ld\n",
      (unsigned long)arg,
      now.tv_sec,
      now.tv_usec);
   iptr[iptr_idx]=kmalloc((unsigned long)arg,
      GFP_ATOMIC);
   if(iptr[iptr_idx] == NULL){
      printk("tasklet: Allocation failed - out of memory\n");
   }
   else{
      memset(iptr[iptr_idx],
         0,
         (unsigned long)arg);
      printk("tasklet: Allocated 0'ed buffer %d (%ld bytes)\n",
         iptr_idx,
         (unsigned long)arg);
      iptr_idx++;
   }
   /* wake up the rt-thread that requested memory */
   set_bit(RTL_SIGNAL_WAKEUP,
      &rt_thread->pending);
}

unsigned long
rtl_kmalloc(unsigned long size)
```

```
{
   int idx;
   pthread_t self = pthread_self();
   RTL_MARK_SUSPENDED (self);
   rtl_printf("rtl_malloc: requesting %ld bytes\n",
      (unsigned long)size);
   /* if we are out of buffer pointers fail without calling the tasklet */
   idx = iptr_idx;
   if(idx < BUFFERS){
      allocator_tasklet.data=size;
      tasklet_hi_schedule(&allocator_tasklet);
      rtl_schedule();
      pthread_testcancel();
      if(iptr[idx] == NULL){
         return -1;
      }
      else{
         return idx;
      }
   }
   else{
      return -1;
   }
   return 0;
}

void *
start_routine(void *arg)
{
   struct sched_param p;
   int ret;
   unsigned long i,size,block;
   p . sched_priority = 1;
   pthread_setschedparam (
      pthread_self(),
      SCHED_FIFO,
      &p);

   pthread_make_periodic_np (
      pthread_self(),
      gethrtime(),
      500000000);

   size=0;
```

```
   block=128;
   i=1;
   while (1) {
      pthread_wait_np ();
      size=block*i++;
      rtl_printf("RT-Thread; requesting %ld bytes of memory\n",
         size);
      ret=rtl_kmalloc(size);
      /* apps must check that they actually got something */
      if(ret == -1){
         rtl_printf("No more buffers available\n");
      }
      else{
         rtl_printf("allocated buffer %d\n",ret);
      }
   }
   return 0;
}


int
init_module(void)
{
   int i;
   for(i=0;i<BUFFERS;i++){
      iptr[i] = NULL;
   }
   return pthread_create (
      &rt_thread,
      NULL,
      start_routine,
      0);
}

void
cleanup_module(void)
{
   int i;
   /* free all non-NULL buffers */
   for(i=0;i<BUFFERS;i++){
      if(iptr[i] != NULL){
         kfree(iptr[i]);
         printk("Freeing buffer %d\n",i);
      }
```

```
    }
    pthread_delete_np (rt_thread);
}
```

3.3.4 Tasklets in RTAI

RTAI provides tasklets for use in RT-context, although the service caries the same name it is not functionally related to the Linux kernel tasklets (although the code and concepts are based on the Linux kernel implementation). The reason for the name selection is that it provides similar functionality and replicates the kernel tasklet behavior for rt-context to a certain extent.

RTAI provides a tasklet API for creation and scheduling as well as manipulation of individual tasklet parameters (i.e. priority). Tasklets are executed before the scheduler is invoked in order of there priority (TODO: phase 2 benchmark influence of tasklets on scheduling jitter). Note that even a low priority tasklet will be executed before the scheduler is called potentially delaying a high-priority task - tasklets thus conceptually are always higher priority than rt_tasks, and should only be used for functions that require short execution times. Tasklet concept in RTAI is intentionally for Defered Service Routines,DSRs, and especially for simple systems that may not even require any rt_tasks, to provide selective functions in rt-context.

Tasklets by default do not safe there FPU register status, so fpu usage must be explicitly requested (a bad idea, as tasklets should run fast), use of FPU in tasklets is not recommended.

The scheduling of tasklets is done by a call to

```
rt_tasklet_exec(tasklet)
```

To assign the function that the tasklet should execute one can pass it at tasklet initialization time.

```
rt_tasklet_init() - initialize a tasklet structure
rt_insert_tasklet() - insert a tasklet in the tasklet list
rt_tasklet_exec() - mark the tasklet for execution
rt_set_tasklet_priority() - change tasklet priority
rt_set_tasklet_handler() - overwrite the handler passed during
                                                rt_tasklet_init
rt_set_tasklet_data() - set the tasklet data field
```

As tasklets by default do not safe fpu registers a tasklet can not use the fpu unless it explicitly requests this resource.

```
rt_tasklet_use_fpu() - announce fpu usage in a tasklet
rt_delete_tasklet() - delete a tasklet fro the tasklet list
rt_remove_tasklet() - delete tasklet in rt-context
```

Timers in RTAI are implemented via tasklets (with two additional time related parameters in the tasklet structure) and are sometimes refereed to as timed tasklets. They have the same execution restrictions with respect to runtime and fpu usage. The management of timer tasklets 'timers' is done via time management task, again execution of timed tasklets happens before the scheduler proper is invoked.

(see section on timers REF)

3.4 sharing memory

Many rt-processes need to share data with non-rt processes or the non-rt Linux kernel. For this purpose the rt-extensions to Linux made use of a shared memory module, mbuff, contributed by Tomas Motylevsky, In this section we are not concerned with this module which is part of RTAI and RTLinux, but rather with sharing memory via mechanisms available from the Linux kernel.

The one way to share memory with rt-space is to add a character device that need not provide more than the open/release and mmap function in the fops (Linux' shorthand for file-operations) and use a kmalloc'ed area that then can be shared, alternatively one can make use of the memory devices in Linux, mmaping /dev/mem.The problem with utilizing `/dev/mem` is that it requires passing the physical address to user-space, the character device is somewhat more complicated but allows clean abstraction of resources.

3.4.1 Simple mmap driver

The simplest method of having shared memory for your RTLinux system is to set up a dummy character device (or drop it into any real device that you need for your system) and provide a mmap call allowing to access a kmalloc'ed area via the mmap system call.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_signal.h> /* RTL_SIGNAL_WAKEUP */
#include <linux/sched.h>   /* flush_signals() */
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>

#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/malloc.h>
```

```
#include <linux/mman.h>
#include <linux/slab.h>
#include <linux/wrapper.h>

#include <asm/io.h>
#include <asm/uaccess.h>

static pthread_t rt_thread;

/* check Documentations/devices.txt for available major numbers ! */
#define DRIVER_MAJOR 17

/* one page - make it page alligned */
#define LEN 4096
static char *kmalloc_area;
```

The rtthread_code is a periodic rt-thread, folowing the typical initialisation part for setting up scheduling parameters (which also can be done in init_module), the thread is marked for periodic execution. The actual runtime code is the lines within thie while (1) ... .

```
static void *
rtthread_code(void *arg)
{
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

   pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

   while (1) {
      pthread_wait_np();
      rtl_printf("RT-Thread current buffer=%s\n",kmalloc_area);
   }
\end{verabtim}

The above {\tt while (1)} loop is conceptually an infinite loop, on exit from the loop c

\begin{verabtim}
   return 0;
}
```

The open method need not do much other than protect the module from being removed by a call to rmmod while it is still in use - this is basically the

minimum open function that will be required.

The close method listed next simply decrements the modules usage count (usecount in `struct module` - see linux/module.h). The kernels module functions check this module count and only permit removal if it is found to be 0.

```
static int
driver_open(struct inode *inode,
   struct file *file )
{
   MOD_INC_USE_COUNT;
    return 0;
}

static int
driver_close(struct inode *inode,
   struct file *file)
{
   MOD_DEC_USE_COUNT;
    return 0;
}
```

The actual function that this device should provide is memmory mapping - the code below will remap the area allocated with kmalloc in `init_module`. As the address base for kernel and user-space are different, the remap must be done using the physical address of the allocated area.

```
static int
driver_mmap(struct file *file,
   struct vm_area_struct *vma)
{
   vma->vm_flags |= VM_SHARED|VM_RESERVED;

   if(remap_page_range(vma->vm_start,
      virt_to_phys(kmalloc_area),
      LEN,
      PAGE_SHARED))
      {
         printk("mmap failed\n");
         return -ENXIO;
      }
   return 0;
}
```

The kernel gains access to driver methods via the file operations which are mapped to major numbers (minor numbers are only differenciated within the driver methods the kernel does not care about minor numbers and just passes them on)

For a device intended for memmory mapping only the minimum fileoperations are open,release and mmap, which are related to the open,close and mmap system calls.

```
static struct file_operations simple_fops={
   mmap:   driver_mmap,
   open:   driver_open,
   release:   driver_close,
};
```

Init module is the place to allocate all resources for real-time systems, even though the driver is in Linux kernel context and thus could safely allocate resources dynamically or even premit swapping of memory to secondary storage, memory for rt-applications needs to be allocated and reserved to allow safe access from rt-context.

The zeroing of memory is not a prinzipal requirement, but the device security policy should tell you if memory needs to be zeroed or not (every device *should* have a security policy...).

Folowing the allocation the character device is registered, basically this will set up a module structure and assign the fileoperation to the requrested major number. User calls to open on the appropriated device file (character device with major DRIVER_MAJOR) will be mapped to the `driver_open` function shown above.

Note that the error handling in this example is incomplete as we do not react to failure of `pthread_create`.

```
static int __init simple_init(void)
{
   struct page *page;
   int ret;
   kmalloc_area=kmalloc(LEN,GFP_USER);
   if(!kmalloc_area){
      printk("kmalloc failed - exiting\n");
      return -1;
   }
   page = virt_to_page(kmalloc_area);
   mem_map_reserve(page);
   memset(kmalloc_area,0,LEN);

    if(register_chrdev(DRIVER_MAJOR,"simple-driver", &simple_fops)
```

```
         == 0) {
      printk("driver for major %d registered successfully\n",
         DRIVER_MAJOR);
      ret = pthread_create (&rt_thread,
         NULL,
         rtthread_code,
         0);
        return 0;
    }
    printk("unable to get major %d\n",DRIVER_MAJOR);
    return -EIO;
}
```

The mandatory `cleanup_module` frees up resources in reverse order of allocation and is called by the kernels module code befor calling the kernel internal module maintenance functions to cleanup internal resources see `kernel/module.c`

```
static void __exit simple_exit(void)
{
   pthread_delete_np (rt_thread);

   unregister_chrdev(DRIVER_MAJOR,"simple-driver");
   kfree(kmalloc_area);
}

module_init(simple_init);
module_exit(simple_exit);
```

3.4.2 Using /dev/mem

The POSIX way of sharing memory is via /dev/mem - you can pass it an offset of 0 and let the kernel select where to place the shared buffer, or you can allocate a buffer and pass the address and size to the user-space side and then use /dev/mem to mmap it to the user-space app. In the given example we simply pass 0 and let the kernel take care of it.

```
#include <rtl.h>
#include <time.h>
#include <rtl_debug.h>
#include <errno.h>
#include <pthread.h>

#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
pthread_t thread;

struct shared_mem_struct
{
   int some_int;
   char ready;
};

int memfd;

#define MEMORY_OFFSET 0

struct shared_mem_struct* shared_mem;

void
cleanup(void *arg)
{
   printk("Cleanup handler called\n");
}

void * start_routine(void *arg)
{
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

   pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);
   pthread_cleanup_push(cleanup,0);

   while (1) {
      hrtime_t now;
      pthread_wait_np ();
      now = gethrtime();
      rtl_printf("I'm here; my shared mem=%d\n",
         shared_mem->some_int);
   }
   pthread_cleanup_pop(0);
   return 0;
}

int
init_module(void)
{
```

```
   int ret;

   memfd = open("/dev/mem", O_RDWR);
   if (memfd){
      shared_mem = (struct shared_mem_struct*) mmap(0,
           sizeof(struct shared_mem_struct),
           PROT_READ | PROT_WRITE,
         MAP_FILE | MAP_SHARED,
         memfd,
         MEMORY_OFFSET);
      if(shared_mem != NULL){
         printk("Dev mem available\n");
      }
      else{
         printk("Failed to map memory\n");
         close (memfd);
         return -1;
      }
   }
   else{
      printk("Failed to open memory device file\n");
      return -1;
   }
   ret=pthread_create (&thread, NULL, start_routine, 0);
   return ret;
}

void cleanup_module(void) {
   pthread_delete_np (thread);
   close(memfd);
}
```

The user space side simply opens /dev/mem an mmaps the offset 0 address.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "device_common.h"
/* device specific defines
 * SIMPLE_DEV = device major number
```

```
 * LEN = shared memory (mmap) buffer length
*/

int
main(void)
{
   int fd;
   char msg[LEN];
   unsigned int *addr;

   if((fd=open(SIMPLE_DEV, O_RDWR|O_SYNC))<0)
   {
      perror("open");
      exit(-1);
   }
   addr = mmap(0, LEN, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

   printf("enter a short test:");
   scanf("%s",&msg);

   if(!addr)
   {
      perror("mmap");
      exit(-1);
   }
   else
   {
      memset(addr,0,LEN);
      strncpy(addr,msg,sizeof(msg));
      printf("Put: %s\n",addr);
   }
   munmap(addr,LEN);
   close(fd);
   return 0;
}
```

3.4.3 Using reserved 'raw'-memory

You can map reserved physical memory by passing the kernel a mem=126m line at the boot prompt (i.e. LILO: for the lilo boot-loader) and then mmap'ing it via /dev/mem (this assumes you have 128m of physical memory installed and want to dedicate 2MB to RTLinux). Not a very elegant way to do it - but a very simple way if you need large blocks of continuous memory. Linux's kmalloc, that provides continuous memory, is limited to 128kB as maintaining a buddy-system

up to 2MB would be a tremendous waste of resources, so continuous memory is limited to de-facto 128kB if you use the Linux kernel memory functions to allocate memory (vmalloc is non-continuous - and not limited to 128kB). There is no need to do any magic for the kernel side to access this area, simply use the physical address of 126*0x100000 as the base address of the 126th MB and manage it on your own.

We do not recomend using this method as it couples application and platform configuration in a verytight w2ay that is not transparent on errors and more or less non-portable.

3.5 non-standard system calls

This is a sample implementation of a system call - system calls are fairly fast compared to device open/read/close operations that need to traverse the VFS and execute a few system calls sequentially, but it is the mosts non-portable and the most dangerous solution to a problem possible, changing a system call or introducing a new one makes your system as a whole incompatible to all other Linux system. Adding a system call can introduce a serious security problem in your system. Adding a system call will require you to patch every kernel release when updating. So the best solution is not to write your own system calls.... but they solve problems some times ;) The actual syscall code is quite simple, and placed in /usr/src/linux/arch/i386/kernel/sys_i386.c for our purposes, naturally if your system call code is more elaborate then you should put it into an independent file.

```
asmlinkage int sys_test_call(void)
{
   /* do something useful in kernel space - like a printk */
   printk("Test System Call called \n");
   return 0;
}
```

This system call will only produce a printk output and thats it - system calls have a fixed number of parameters and types that must be declared, in the above case the system call takes no arguments at all. The number of arguments not only needs to be given with the declaration of the system call but also with the prototype declaration which is a little bit different than regular prototype declarations (see below).

The kernel has a "jump-matrix" for the system calls - the position of a system call in the syscall table is absolute so you can't add in your system call at the beginning or in the middle or your will break the entire system, if at all add it at the end of the syscall table. The position in the syscall table is the syscall number. So put it into the syscall table like:

/usr/src/linux/arch/i386/kernel/entry.S

```
...
.long SYMBOL_NAME(sys_getdents64)    /* 220 */
.long SYMBOL_NAME(sys_fcntl64)
.long SYMBOL_NAME(sys_test_call)
```

Note that this system call table may change over time - so you will have to patch newer kernels with your system call and modify the code that is calling the syscall since the number may have changed - it is up to you to maintain your system call.

If you want to put your syscall at a position beyond the last current system you must fill up the system call table with empty system calls:

```
.long SYMBOL_NAME(sys_ni_syscall)
```

After recompiling your kernel you could now call it with the absolute system call number, to be a bit more user friendly you need to add some entries to make it available to user space apps via asm/unistd.h:

```
/usr/include/asm/unistd.h
#define __NR_test_call      222   /* this number better be
                 the same as the position in entry.S !! */
```

Now a regular system call like open is simply called by
fd=open(".....

- our system call could also be called in this way but that would require recompiling glibc as well, as during the build process of glibc the kernels syscall table is read - if you do recompile glibc then you have reached the maximum possible incompatibility to any other Linux system. If you don't want to recompile glibc, which is probably a good idea, then you need to put the prototype declaration for your system call into the source file.

So assuming we did not recompile libc, call it in in a c-source file like:

```
#include <asm/unistd.h>
#include <errno.h>

_syscall0(int,test_call);

main(){
   syscall(222); /* call it via syscal(SYSCAL_NUMBER) */
   test_call();  /* call it by name */
   return 0;
}
```

Compile with simple gcc syscall.c -o syscall and run this program as ./syscall. To check kernel output (the printk that our syscall is to do) use the dmesg command - it should have produced:

Test System Call called Test System Call called

- the two calls are via syscall(222) and test_call() - note that you don't need the headerfiles errno.h and asm/unistd.h to use syscall(222) but you do need these includes for the named call test_call(); Using syscall(222) can be very confusing as it says nothing about what you are trying to do, so give application specific system calls a meaningful name. Further it should be noted that modifying the system call layer requires that these changes are well documented in the context of the modified kernel, the system call layer does not change with every kernel release but it does change from time to time, so it is insufficient to only document the application specific system call mechanism but all modified kernel files need to be included.

One possible limitation to the application specific system call is that this is a modification to the kernel core, which is under GPL, thus such modifications that can hardly count as utilizing the normal kernel interfaces (under which Linus Torvalds permits LGPL licensing) is also under GPL license.

3.6 Shared waiting queue (Experimental)

The shq package that is currently external to RTLinux-3.2-preX (Linux-2.4.18) provides a basic mechanism for synchronizing RTLinux tasks and Linux kernel threads. It permits suspending RTLinux threads and Linux kthreads waiting for common events. Signaling between rt and non-rt kernel context can be done with the existing RTLinux/GPL API (see examples/kernel_resources in the RTLinux-3.2-preX releases for details) but this does not provide wait-queue facilities in an rt-safe way to sync on specific events (that is currently application programmers must build there own infrastructure for synchronization). The shared waiting queue type,`shq_wait _queue_t`, and functions for job control are provided.

3.6.1 shq API

Non POSIX API, self defined, as shared wait queues are non-POSIX them selves.

```
shq_wait_init() - initialize a shared wait queue
shq_wait_destroy() - destroy a shq_wait_queue_t type wait queue
shq_wait_sleep() - a function to suspend the current job
shq_wait_wakeup() - to wakeup jobs waiting in a queue
```

3.7 Accessing kernel-functions

TODO: limitations of non-atomic access
blocking access
potential priority inversion problems
how to know which are safe -¿ part2 analysis of rt-safe kernel functions.

Chapter 4

RT/Kernel/User-Space Communiction

In this section we scan the available mechanisms for communicating between RT-context, Kernel-space non-RT-context and user-space.

4.1 Standard IPC

There is no one standard for interprocess-communication, but there are a number of standards involved, in this section we discuss IPC mechanism that are based on some standard, not necessarily on a POSIX standard.

The IPC mechanisms introduced here are what one would typically expect to be available in any RTOS, Linux based or not, and all of these mechanisms are provided by all of the implementations in the one way or other, note though that not all implementations may follow standards, or may only follow standards if options are restricted.

Splitting synchronization and IPC is not always done in computer science literature, as theses are strongly related, under IPC in this document we will list all mechanisms that provide data exchange along with the necessary synchronization.

4.2 Synchronization objects

Obviously no RTOS can live without synchronization objects. Available sync objects are listed below. The first three in the list are standardized and not described here in any further detail (the actual support of the standard may varies though, and is notably not cleanly supported in RTAI). Bits, a non-standard extension in RTAI, and global variables , need some additional notes.

- semaphores
  Supported in all variants: RTAI provides an additional, so called, typed

semaphore (counting,binary and resource semaphores). Some additional conditional waiting modes (rt_sem_wait_if,rt_sem_wait _until,rt_sem_wait_timed), note that RTAI does not use the POSIX syntax for semaphore destruction, sem_destroy, but rather rt_sem_delete.

- mutex
  Supported in all variants: mutex related priority inheritance is supported in RTAI and RTLinux/GPL (unclear on support in RTLinux/Pro - no docs). There is no documentation for the RTAI mutexes in the official RTAI user-manual or in the documentation coming with RTAI releases.

- conditional variables
  Supported in all variants: (TODO: check details of POSIX compliance in RTAI). No documentation for condvars in RTAI available.

- barriers
  Supported in RTLinux/GPL only: It is experimental in version 3.2-pre3 (external patch expected to be merged before 3.2 final). (Note the pthread _barriers in RTAI are listed in some documents but are not available in RTAI as of version 24.1.11 - status of pthread_barriers is currently unclear in RTAI)

- spinlocks
  Supported in all variants: RTLinux/GPL and RTLinux/Pro support POSIX pthread_spinlock functions, RTAI utilizes the Linux kernel spinlock functions (they are patched though in the RTHAL kernel patches). Spinlocks on all versions (and in Linux) on UP systems reduce to disabling/enabling interrupts.

- bits (flags)
  Supported in RTAI only: bits in RTAI provides functions for creating compound synchronizations objects based on AND/ORs on a 32bits 'flags' variables. Bits provide a counter whose state depends on the combined flag variables, it behavior is comparable to a counting semaphore just without the notification functionality (signaling) and waiting functions.

- global variables Obviously all variants support global variable sharing as they all operate in kernel address space when running in kernel-mode, for user-space RT implementations this is naturally not the case (true for PSC,LXRT and PSDD).

The problem we see with the bits extension to RTAIs synchronization objects is not only that it follows no standardized mechanism, but that the provided service has no formal specification and no method of assessment associated. A non-standard extension is ok if a clean specification is provided that allows

validation, beyond that we seriously question that a formal analysis of a task-set utilizing bits functionality can be done, therefore we don't recommend using this facility.

Global variables are a great way of producing unmaintainable code and definitely non-portable code, we recommend that global variables, if used, be limited to the scope of a source file (that is declared static), exporting of variables to global kernel context should be done with care as to prevent name-space pollution in the Linux kernel.
To eliminate problems of name-space collision (although recent kernels manage that ok) global variables should follow a global naming convention that may be project specific, our recommendation is to prepend the module name to the variable to make them easy to associate with appropriate modules. Usage of global variables should be limited and are no replacement for shared memory (as sometimes done).

4.2.1 FIFO

All variants of hard-realtime extensions to Linux provide RT-safe FIFIs to communicate between user-space and RT-context as well as between RT-processes. Beginning with the mailbox implementation in RTAI the FIFO mechanism is no longer considered the primary IPC mechanism, though full backwards compatibility is maintained to the original RTLinux (NMT) FIFO implementation. RTLinux/GPL and RTLinux/Pro continue to support native FIFOs. All implementations allow preallocating FIFOs, allowing opening and closing in RT-context. As FIVOs are required to be non-blocking in RT-context when transferring to non-RT context the issue of managing overflow arises, this is currently 'solved' in all cases by simply discarding data if the FIFO overflows, thus it is up to the programmer to check/verify data integrity/completeness.

RTAI provides additional synchronization objects based on FIVOs - refereed to as rtf_sem functions, allowing to share semaphores between RT and non-RT context via RT-FIVOs.

Basic API functionality provided in all variants

- create (allocation, allowing to define the size)
- open
- resize
- put/write
- get/read
- close
- destroy

- assignment of handlers (callbacks for read/write operation)

RTAI

RTAIs implementation was based on the RTLinux version up to the 1.X RTAI releases, with the introduction of mailboxes, the RTAI FIVOS use mailboxes to 'emulate' FIFO behavior (TODO: phase 2 benchmark overhead of emulation). Although no longer natively implemented in RTAI 24.X releases the original RT-FIFO API is still available for backwards compatibility. For new developments the use of FIFOs is not recommended, but clearly the native-supported mechanisms (mailboxes, message queues) should be utilized.

In addition to the API listed above RTAI provides some extensions:

```
rtf_reset - flush the content
rtf_write_timed - write with timeout (user-space)
rtf_read_timed - read with timeout (user-space)
rtf_read_all_at_once
rtf_suspend_timed - let user-space app sleep for a delay
rtf_set_async_sig - send SIGIO on data
```

A further set of extensions to the FIFO API for sharing semaphores between kernel and user-space is provided via the `rtf_sem_` set of functions, this sounds like a great way to cause priority inversion problems, use of this facility requires especially careful design.(TODO: phase 2 benchmark overhead of synchronization, validate concept)

The POSIX wrapper layer provided in the POSIX compatibility module is note listed here as it is incomplete and does not provide the functionality the non-POSIX API provides. For RTAI applications we recommend using the non-POSIX API as this is the native API.

RTLinux/GPL

RTLinux/GPL continues use of the original RT-FIFO implementation, but provides a POSIX extension to allow POSIX compliant read/write instead of rtf_get/rtf_put. It is to be expected (as announced by FSMLabs Inc.) that the native POSIX-FIFO implementation recently releases in RTLinux/Pro (Version 1.2) will be merged into RTLinux/GPL . The RTLinux/PGL implementation currently requires non-POSIX operations at initialization time and at FIFO removal (rtf_create, rtf_destroy respectively). For pure POSIX compliant code, preallocated FIFOs must be used.

To avoid the non-POSIX calls for creation and deletion the calls to open in non-RT , that is Linux init_module context need to pass the O_CREATE flag.

- preallocated - open("/dev/rtf0", O_CREATE—O_NONBLOCK);

- dynamic - created open("/dev/rtf0", O_NONBLOCK); (rtf_create was called BEFORE the call to open)

The POSIX compliant access to the RT-FIFOs is currently provided via a POSIX compatibility module `rtl_posixio.o`, the native implementation is the non-POSIX API. This non-POSIX function set provided creation of RT-FIFOs, assignment of a handler to a RT-FIFO that is called on receiving and transmitting data. Fifo handlers are availalbe in two variants, rt_handlers trigger on read/write from RT-context, or regular handlers, triggert on writes from user-space. De-facto this allows signaling from user-space to RT-context via RT-FIFOs, in the opposite direction the standard UNIX blocking IO functionality allows signaling (i.e. select). As FIFOs by design are in prinzipal uni-directional the API provides wrappers for creation of bi-directional FIFOs, joining two FIFOs to a paired FIFO that can be accessed via a single FIFO-number in read-write mode on both ends.

```
rtf_create - create a FIFO
rtf_create_handler - assign a user-space trigger handler
rtf_create_rt_handler - assign an RT-space trigger handler
rtf_make_user_pair - create a bi-directional FIFO
rtf_link_user_ioctl - link user-provided ioctl function
rtf_destroy - remove a RT-FIFO
```

The non-POSIX functions for data and status management in RT-FIFOs

```
rtf_get / rtf_put
rtf_flush
rtf_isempty
rtf_isused
```

A POSIX wrapper layer providing

```
open
read
write
close
```

is available. As the future of RTLinux/GPL FIFOs is to move towards a pure POSIX we recommend using the POSIX compliant API as far as possible (the only exception being the required rtf_create() / rtf_destroy call on non-preallocated FIFOs), and the use of preallocated FIFOs where possible.

RTLinux/GPL will though maintain backwards compatibility with older API versions but this may injure a processing overhead and thus performance decrease.

RTLinux/Pro

As RTLinux/Pro currently does not provide mailboxes or message queues, the primary method for communicating between RTLinux/Pro threads and Linux processes are RTLinux FIFOs. These have been limited in the original implementation by there use of predefined names `/dev/rtf0` - `/dev/rtf63`. With release of RTLinux/Pro 1.2 creation of RT-FIFOs is no longer limited to a specific path/name convention but follows regular UNIX semantics, allowing dynamic creation of FIFOs within the GPOS [47].

The RT-FIFOs provide

- dynamic creation
- synchronous IO
- asynchronous IO

Backwards compatibility to the old interface using `/dev/rtfX` is supported though.

A POSIX compliant creation and use of a RT-FIFO in RTLinux/Pro would look like the following code sample, note that mkfifo must be called from non-RT Linux context unless the required buffer space was preallocated. Preallocation of FIFO-buffer space is provided at compile time via the configuration menu.

```
int init_module (void)
{
   int fd;

   if ( (fd = mkfifo("/tmp/myfifo", 0)) )
      return -1;

   if ( (fd = open("/tmp/myfifo",
            O_RDONLY|O_NONBLOCK)) >= 0 )
      return -1;

   close(fd);
   return 0;
}
```

The FIFO API in RTLInux/Pro:

```
open(2)
close(2)
write(2)
```

```
read(2)
lseek(2)
ioctl(2)
unlink(2)
mkfifo(3)
```

NOTE: the (2),(3) behind the function names refere to the standard manual pages for a full documentation of the function syntax.

`mkfifo` has some RTLinux/Pro specific behavior, if the call to mkfifo is done with the file permission set to 0 as shown above then it will only be visible in RTLinux but not in Linux context, To make it visible in Linux the permission field must be non-zero a

```
mkfifo("/myfifo", 0755);
```

A somewhat dangerous behavior of mkfifo is that there is no error reported if the filename passed to `mkfifo` already exists, in that case the file is simply removed (!) and recreated as a RT-FIFO. Considering that the operation of inserting a kernel module requires root-privileges this seems like a bad design decision. As long as this behavior is the default, and no error reporting is included, the new style FIFOs can't be recommended (TODO: figure out what the rational behind this design decision is...).

An extended non-POSIX function set is also provided for creation of RT-FIFOs, of interest is the ability to associate a handler with a FIFO that is called on receiving and transmitting data, such handlers can be installed for user-space writes as well as for inter-task communication in RT-context. De-facto this allows signaling from user-space to RT-context via RT-FIFOs.

For creation of bi-directional FIFOs two FIFOs can be coupled to a paired FIFO that can be accessed via a single FIFO-number.

```
rtf_create - create a FIFO
rtf_create_handler - assign a user-space trigger handler
rtf_create_rt_handler - assign an RT-space trigger handler
rtf_make_user_pair - create a bi-directional FIFO
rtf_link_user_ioctl - link user-provided ioctl function
rtf_destroy - remove a RT-FIFO
```

The non-POSIX functions for data and status management in RT-FIFOs

```
rtf_get / rtf_put
rtf_flush
rtf_isempty
rtf_isused
```

Although the non-POSIX FIFO extensions provide some unique features, it is not recommended to utilize these non-POSIX concepts in new projects. It can be expected that the support for non-POSIX extensions will be provided long-term for backwards compatibility but the native implementation will be (or de-facto already is) the POSIX compliant API, which may inflict on the performance of the non-standard API.

4.2.2 SHared Memory, SHM

Along with the development of RT-FIFOs as a means of communicating data between RT-context and user-space, shared memory was provided fairly early in the development of hard real-time enhanced Linux extensions. The original mbuff module supported RTLinux as well as RTAI.

(TODO: phase 2 compare bandwidth of shm and influence on RT-performance when heavily used in user-space).

mbuff

Mbuff is expected to fade out with the availability of the new POSIX compliant layer in RTLinux/GPL, both RTAI and RTLinux/Pro already implement an mbuff independent shared memory subsystem.

The API of mbuff was a non-POSIX alloc/free equivalent. The issue of passing the address from kernel to user-space was solved by creating a dedicated device `/dev/mbuff` from which the addresses of the specific section could be retrieved.

RTAI

The RTAI shm functions, refereed to as SHM service functions, provide a sysV SHM alike concept of named memory areas but a malloc/free like API for access of data areas. Additionally some name management functions are provided for mapping of addresses to names and back.

```
rtai_malloc - user-space
rtai_malloc_adr - user-space to dedicated address
rtai_kmalloc - RT-space (kernel-space)
rtai_free - user-space
rtai_kfree - RT-space (kernel-space)
```

The service functions for converting names to numeric identifiers used internally and vice-versa are:

```
nam2num - convert name to numeric
num2nam - numeric to name
```

shared memory areas can be accessed from user-space, from LXRT user-space realtime and from kernel-space with a common API set.

For users-space usage the functions are provided as inline-functions via the `rtai_sh.h` header file.

Some additional (not-documented) API extensions for status management of shared memory areas is available (may be incomplete)

```
rtai_check - check if the name exists
rtai_is_closable - return closable value
rtai_not_closable - set closable to 0
rtai_make_closable - set closable to 1
```

The shm functions use the sysrequest (srq) facility which are software interrupts to signal from kernel to user-space. (TODO: benchmark effects of heavy user-space usage of sysrequests on RT-performance)

RTLinux/GPL

RTLinux GPL currently only provides the mbuff module for shared-memory, this will change in the near future though.

```
open
mmap
munmap
ioctl
close
```

The management of status and name/region information in mbuff is done via ioctl system calls on the `/dev/mbuff` device file. This device file has no official major/minor number assigned it currently uses major 254 which is for experimental device usage.

RTLinux/Pro

The shared memory implementation in RTLinux/Pro follows the POSIX standard strictly. The shared memory devices are created dynamically when open is called on them via the `shm_open` function. This provides a file-descriptor for accessing the newly created device, which then is resized via the ftruncate system call.

After that it can be mmap'ed just like a regular file or device file in user-space, use in RT-context requires that the open be called in non-RT context, `init_module` which is executed in Linux kernel-context, the unlink must be called in cleanup_module of the realtime kernel module .

Shared memory creation and destruction functions

```
shm_open - create a shared memory device
shm_unlink - destroy it
```

The mapping follows the strict POSIX api, note although the actual allocation happens in the ftruncate function this MUST be called from non-RT context.

```
mmap
munmap
ftruncate - resize it
ioctl
close
```

This facility is also accessible from PSDD, user-space real-time applications.

4.2.3 ioctl/sysctl

ioctl are supported on RT related devices (RT-FIFOs in all implementation and on memory devices in RTLinux/Pro, sysctl functions can be interfaced to RT-context for kernel-mode RT processes. For use of sysctl in relation to RT-processes see the section on `/proc` and sysctl below.

RTLinux/GPL and RTLinux/Pro allow assigning of user-provided ioctl functions to the associated RT-FIFOs, this is accomplished through the non-POSIX `rtf_link_user_ioctl` API extension. Ioctl extension are only supported for calls from user-space context, but not from within RT-context (where they make little sense any way as in a flat address space direct access to all internals is given any way).

Sysctl functions are not supported in any RT-specific way, that is non of the implementation offers a pre-defined module for accessing sysctl facilities in the Linux kernel, but all implementations can utilize this standards compliant kernel resource, see section on accessing kernel resources for details.

4.3 Implementation specific standard IPC

Message queues and Mailboxes are conceptually similar to FIFOs but they add meta-data to the byte-stream processed by the mailbox/message queue. FIFOs are simply a byte stream, what goes in one end comes out the other in the same order, mailboxes and message queues, deliver the byte stream in arbitrary size chunks of data and associate a ‘header‘, that can be viewed as an envelope, with each message.

The main distinction between message queues and mailboxes is that message queues are non-copying whereas mailboxes copy the data.

4.3.1 RTLinux/GPL message queues

The RTLinux message queues are currently an external package (it is to be expected that it will be merged as soon as it is found stable and has reached its final implementation ??cera_mqueues)

In RTLinux, the most flexible IPC mechanism available is shared memory ??buff available as mbuff, in that case though its programmers responsibility to use appropriate synchronization mechanism to implement a safe communication mechanism between RT-threads. On the other hand, signals and pipes lack certain flexibility to establish communication channels between RT-threads.

In order to cover some of these weaknesses, POSIX standard proposes a message passing facility that offers:

- Protected and synchronized access to the message queue. Access to data stored in the message queue is protected against concurrent access.
- Prioritized messages. Processes can build several flows over the same queue, and it is ensured that the receiver will pick up the oldest message with highest priority.
- Asynchronous and timed operation. Threads don't have to wait for send/ receive completion (non-blocking), i.e., they can send a message without having to wait for someone to read that message. They also can specify a timeout (mq_timedsend/mq_timedreceive) if the message queue is full/empty, to wait until returning failure.
- Asynchronous notification of message arrivals. A receiver process can configure the message queue to be notified on message arrivals.

The POSIX message queue implementation for RTLinux is currently an external module (not yet extensively tested and thus not yet merged into the main tree) it is to be expected that it will be merged in the near future.a

POSIX message queues copy data from the sender to the receiver, and require that POSIX signals and timers be configured (scheduler is a bit slower with these configured than without).

It is preferable to use POSIX messages queues to communicate prioritized data between RT-threads and not use FIFOs and 'home-brew' synchronization. As POSIX message queues have just appeared in RTLinux-3.2-preX there is no backwards compatibility issue involved, forward compatibility can be expected as the implementation targets POSIX compliance.

4.3.2 RTLinux/GPL POSIX signals

The RTLinux/GPL POSIX signals are integrated into the main development as of RTLinux-3.2-pre2. Currently RTLinux/GPL is the only hard real-time variant that implements POSIX compliant signals and an appropriate test-suit.

POSIX signals are delivered at the end of the schedulers execution, that is after a task was selected for execution the pending signals are tested and delivered.

POSIX signals management increases the scheduling execution time and thus is provided as a compile time option. The POSIX timers depend on the POSIX signals facility in RTLinux/GPL.

4.3.3 RTAI message queues and mailboxes

It is hard to say where this section should go as RTAI implements a number of functions that are based on standardized concepts but don't follow these standards, this is simply because RTAIs maintainers try to optimize the system and don't make any performance compromizes to standardization, and furthermore they like to add features requested or provided by users, to overcome some of the limitations that standards compliance would enforce...

RTAIs mailbox implementation originally used a FIFO which allowed arbitrary sized messages, but delivery was in FIFO order. recent extensions have added typed mailboxes that allows to add a receive/transmit policy to the messages:

- unconditional: block until message was delivered
- best-effort: pass as much as can go without blocking
- conditional: pass the entire message if possible without blocking otherwise fail.
- timed: timeouts for delivery/receive (absolute or relative)

This in our opinion is a good example of RTAIs policy of extending the capabilities but we clearly question the realtime compliance of this approach as it is hard to design appropriate exit strategies in hard-realtime application to allow such failure or partial failure cases. The issue is that RTAI while enhancing the capabilities extensively puts a large burden on the application programmer/designer and open a number of pitfalls for applications. Typically the effects of such policy extensions will be hard to test (i.e. how to test a 'best-effort' mailbox to determine worst-case performance ??) and validation of such designs becomes complex - this is not to say they are bad or use-less, but it should be stated that the limitations that the standard conform implementations impose are very well considered and really are inherent limitations especially for realtime systems.

We consider these non-standard extensions problematic as they don't come with an appropriate test-suite and underlaying design guidelines, thus the probability of falling into pitfalls that are related to these extensions is quite large especially for programmers that don't have a well established Linux kernel and hard-realtime background.

As RTAI provides services called message queues and mailboxes as well as typed mailboxes and these all are somewhat comparable to POSIX message queues we treat them all in this section.

RTAI message queues

A message queue can be seen as a FIFO with a header per data item added, aside from the meta-information added there is no real difference. message queues can be seen as character devices, although they have no formal device file associated with them.

The RTAI message queues are not to be confused with the POSIX message queues, the RTAI mailboxes follow the semantics of POSIX message queues, not the `mq` in RTAI !

Message queues in RTAI don't copy the data for delivery, that is the sender puts the data in the queue, signals the receiver and the receiver retrieves it from the same memory location, there is no intermediate copy operation (which POSIX message queues and RTAI mailboxes do).

The `mq` API is a send/receive type API:

```
rt_send - blocking send
rt_send_if - only send if receiver is not blocked
rt_send_until - blocking send - block until abstime
rt_send_timed - blocking send - block relative time
rt_receive - blocking receive
rt_receive_if - only receive if non-blocking
rt_receive_until - blocking receive - absolute time
rt_receive_timed - blocking receive - relative time
```

The actual queuing policy of blocked tasks is either in priority order or in FIFO order, this is not configurable with the configuration tool `make menuconfig`, and the scheduler variable MSG_PRIORD that is given in the documentation for setting the behavior in the scheduler source code is not defined (looks like the documentation is kind of out of date..), code inspection indicates that priority order is implemented in the `enqueue_blocked`, function and that there is no code to provide a FIFO order.

TODO: benchmark message queue throughput and management overhead (sync overhead)

RTAI mailboxes

RTAI mailboxes are closer to what POSIX referees to as message queues than RTAI message queues. RTAI mailboxes provide delivery modes for

- unconditionally - block until message is delivered/received

- unconditionally but only pass the bytes that can go without blocking
- conditional delivery if the whole message can be passed without blocking
- blocking with timeout, timed absolutely or relatively
- overwriting form of send is also available, useful for logging.

These modes are available for sending and receiving (except the last mode that is obviously only for senders).

The mailbox implementation allows for 'fragmented' delivery, that is the send buffer may be smaller than the message size, in which case multiple send operations are invoked to deliver a single message (TODO: benchmark fragmentation behavior), to our understanding this fragmenting feature is inherently bad for a realtime system and can not be recommended for hard real-time systems, for soft real-time systems this may be an option to reduce resource demands.

Mailboxes in recent release are wrappers to the types mailboxes, though the native send/receive functions are called (only the init is actually a wrapper).

The mailbox API can be used symmetrically in RTAI kernel and LXRT user-space.

```
rt_mbx_init
rt_mbx_delete
rt_mbx_send
rt_mbx_send_wp - send as much as possible without blocking
rt_mbx_send_if
rt_mbx_send_until
rt_mbx_send_timed
rt_mbx_receive
rt_mbx_receive_wp - receive as much as possible without blocking
rt_mbx_receive_if
rt_mbx_receive_until
rt_mbx_receive_timed
```

Note that there is no sending function provided for the overwriting behavior listed in the documentation, from source code inspection we conclude that this feature is not available as of RTAI 24.1.11 .

RTAI Typed mailboxes

The RTAI offers extended mailboxes refereed to as TBX for Typed Mail Boxes (configurable at compile time).Typed mailboxes (TBX) are an alternative to the default RTAI mailboxes with the additional features of:

The API for typed mailboxes is identical to the one for regular mailboxes (see above), as of RTAI 24.1.11 the actual implementation is based on the concept of typed mailboxes (TBX).

- Message broadcasting, that means send a message to ALL the tasks that are blocked on the broadcasting TBX. (TODO: check wakeup behavior)
- Urgent sending of messages: these messages are not enqueued, but inserted in the head of the queue, bypassing all the other messages already present in TBX.
- The PRIORITY/FIFO wakeup policy can be set TBX creation.

Features like the `Urgent sending`, Last In First Out (LIFO) delivery order for an individual message, are not recommended, this is an example of sloppy management of priorities, we also see it as a serious limitation that features like this can't be validate in the context of an application, basically this sounds like a great way to cause implicit priority changes resulting in hidden priority inversion problems.

TODO: benchmark mailbox throughput and management overhead (sync overhead) as well as priority related aspects.

4.3.4 non-standard IPC

Standard IPC mechanisms were designed for user-space, this has some implications for the hardrealtime extensions operating in kernel space.

- no provisions to communicate with non-rt kernel processes
- the IPC mechanisms are not designed RT-safe for IPC between RT-processes
- optimized, Linux-specific methods, are not available

In this section we introduce a few of the non-standard IPC mechanisms available to RT-processes in kernel-space, not that user-space realtime also utilizes some of these facilities as user-space realtime is de-facto a 'user-space kernel-module', that is user-space realtime from the standpoint of IPC can be treated as kernel-context in many respects.

Kernel messages

Within the Linux kernel messages produced by kernel functions are queued up in a 64k (default size) ring buffer which is then extracted by a user-space application `klogd/syslogd` and sorted into log-files, this facility for error reporting is made available to the realtime extensions via non-POSIX functions rt_printk and rtl_printf.

All realtime kernel extensions offer a simple mechanism for pushing messages out from a kernel module to the kernels message ring-buffer. rt_printk (RTAI) and rtl_printf(RTLinux) is a RT-safe printf call that exists within the

realtime kernel, and works the exact same way as printf or printk, but is made safe from a realtime process by disabling interrupts, appending the message to the kernels ring buffer, and reenabling interrupts. This method is very useful but should be used with care due to the conceptually expensive mechanism involved. rt_printk/rtl_printf should not be used in production code as status report facility but only for error reporting - its primary usage is during code development. **Note:** although rt_printk/rtl_printf is frequently used in non-RT context (especially init_module and cleanup_module) this should not be done as these functions will disturb realtime performance (see note on interrupt disabling ??nterrupt_disabling)

To put it in the words of the RTAI developers
`rtai-24.1.11/rtaidir/rt_printk.c`:

```
/* Latency note: We do a RT_spin_lock_irqsave() and then
 * do a bunch of processing.  Is this smart?  No.
```

The notification mechanism to synchronize the RT-executive with the non-RT Linux kernel for message appending to the ring-buffer, is implemented via virtual, or soft, interrupts in RTLinux. RTAI uses a reimplementation of the kernels printk function, directly memcpy'ing to the kernels ring-buffer. RTAIs implementation allows configuration of the rt_printk's buffer size at compile time.

Direct writing to the console is also supported in all hardreadltime variants by providing wrappers to the kernel console functions, this is very usable as it allows printing to the console even if Linux freezes (by the RT-executive using up the full processing power) or is shut-down, but it introduces substantial latency in RT-context and is also primarily for debugging and fatal-error reporting.

device drivers

A means of building simple and flexible, but still POSIX-conform shared resources is to put them into device drivers. Linux device drivers can be pure software devices, see Rubinis sample devices [3]. With such dedicated device drivers it is simple to provide the user-space with standardized system call-interfaces and allow the RT-side operating in kernel-space to directly access the shared resource.

The following example simply provides functions in kernel space that can be called from user-space and kernel-space that only execute a print statement , replacing this print statement with an application specific service is all that conceptually is required. It should though be noted that a project should provide a security policy for designing device drivers as they operate in kernel space and are thus security critical components of an OS/RTOS.

```
static int
driver_open(struct inode *inode,
```

```
   struct file *file )
{
    printk("driver open called\n");
    return 0;
}

static int
driver_close(struct inode *inode,
   struct file *file)
{
    printk("driver close called\n");
    return 0;
}

static ssize_t
driver_read(struct file *File,
   char *buf,
   size_t count,
   loff_t *offset)
{
    printk("driver read called\n");
    return 0;
}

static ssize_t
driver_write(struct file *File,
   const char *user,
   size_t count,
   loff_t *offset)
{
    printk("driver write called\n");
    return 0;
}
```

Note that the basic framework here is not in any way real-time specific , it simply is a regular Linux device driver, but it provides a means for user-space applications to gain access to kernel functions via POSIX compliant system calls.

```
static struct file_operations simple_fops={
   THIS_MODULE, /* need this only for 2.4.X kernels */
    NULL,  /* llseek */
    driver_read,  /* read */
    driver_write,  /* write */
    NULL,  /* readdir */
```

```
    NULL,  /* poll */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    driver_open,  /* open */
    NULL,  /* flush */
    driver_close,  /* release */
    NULL,  /* fsync */
    NULL,  /* fasync */
    NULL,  /* lock */
};

static int __init simple_init(void)
{
    if(register_chrdev(SIMPLE_MAJOR,DEV_NAME, &simple_fops) == 0) {
        printk("driver for major %d registered successfully\n",
          SIMPLE_MAJOR);
        return 0;
    };
    printk("unable to get major %d\n",SIMPLE_MAJOR);
    return -EIO;
}

static void __exit simple_exit(void)
{
    unregister_chrdev(SIMPLE_MAJOR,DEV_NAME);
}

module_init(simple_init);
module_exit(simple_exit);
```

The registration and unregister functions are somewhat kernel version specific the ones shown here are for the 2.4.X series of Linux kernels, but the basic structure of such a module will hardly change substantially in the near future.

For more examples of coupling application specific device drivers with RT-executives, see `examples/kernel_resources/` in the current RTLinux/GPL release (although as noted the core framework is quite variant independent), a fairly complete documentation is also available for these examples [**?**]

kernel threads

see section on kernel resources for details and implementation notes/examples

/proc filesystem

The /proc interface is a well established and widely used interface in the Linux kernel, beginning with the late 0.99.X releases of the Linux kernel it has been part of the official kernel releases. First versions focused on network issues but additional subsystems quickly began using proc files to simplify administrative and debug tasks. With the early releases the API was fairly complex as of Linux kernel 2.4.X the API for the proc interface is very user-friendly. The main features of the proc filesystems summarized:

- Direct access to kernel internals
- Simple API
- Simple access via filesystem-abstraction
- POSIX compliant open/read/write/etc. Interface
- Kernel level security setting on a file-scope

In this section a introduction to the proc interface specifically for embedded an real-time Linux is given, the concepts are applicable to all flavors of realtime enhanced Linux, the examples shown are based on RTLinux/GPL though. Work on this type of interface is an on-going GPL effort at OpenTech Research, Austria citeproc-utils. For the details and specifics of building an interface using the proc FileSystem see citeembedded-proc. Here we only give a basic concept overview of this special FileSystem is given. Proc FileSystem entries are not stored on a non-volatile media like a harddrive, they are generated on the fly, that is every time that the read-method of the associated file is invoked. This give a very large freedom in the way output is represented to the user without requiring to parse complex input formats just to stay user-friendly. The proc-filesystem is a filesystem in the sense that it provides a interface to user-space that resembles a normal VFS interface of any other filesystem allowing POSIX style access.

The two basic interface types in proc are character based text-mode interfaces and binary interfaces - most are text-mode and in the cases where binary interfaces are used you normally have both implemented as it is simpler to interface user-space apps to binary interfaces than to text-mode interfaces that would require parsing (or at least scanf'ing fixed format input lines), but the binary interfaces are not well suited for direct interpretation by humans. As an example /proc/pci and /proc/bus/pci/devies basically contain the same information, just one interpreted and the other raw.

4.3.5 Performance

The main reason to actually start playing with the `proc` interface is performance of some standard Linux tools - running applications like `top` or the ps-utilities on embedded systems showed that these tools simply had to high CPU-demands for the system. Analyzing why this is so showed:

- System calls are expensive - and heavily used by some tools
- The executables are large because they are providing too much
- FileSystem utilization issues arise if you build many small tools of your own (well busybox could solve that...but)
- Not everything we wanted to see was accessible easily

Lets look at some of these issues in a bit more detail as I think they could be relevant for the analysis of other performance bottlenecks on embedded systems.

System calls

System calls are the preferred, standardized and safe way to cross the kernel-user-space boundary. But they are expensive if heavily used. A simple `./hello_world` performs about 30 system calls, `echo "beep"` is up to 42 (numbers may verier a bit on different systems (kernel and glibc dependant)), so that is about the bottom line for more or less any user space application. But looking at some of the typical admin tools like top makes it clear - top takes up to a few thousand system calls to build the output for a single page (SuSE 8.0 default installation) - and the default is to update the content once per second - so on a reasonably reduced system top still can cause one thousand system calls to output a single page. But basically it is only collecting information that is stored entirely in the Linux task structure - so running through this task structure in the proc read method and outputting it in a top like for to the console only takes one system call more than `echo beep` did !

Optimized File OPerationS, fops possible

A general FileSystem layer provides a POSIX type open/read/wrtie/close/etc. interface to the programmer - the data blocks are a general data abstraction, which is very flexible but suboptimal if the data is very specific and especially small data amounts. The `/proc` FileSystem has a different approach - the file operations are split in FileSystem specific open/release and not FileSystem specific but file specific read/write, allowing to optimize them not only with respect to performance but also the data representation. In the examples given later we see how to register a specific read/write method that allows to present kernel internal or driver specific data structures in a formated manner as well as perform data interpretation within the read/write methods.

FileSystem overhead

General purpose FileSystem have a certain overhead, management objects like inodes/superblocks are required to interface to the operating system and data-block are discreet leading to fragmentation effects. The proc FileSystem can build application/problem specific data-'blocks' and thus optimize the FileSystem layer minimizing memory usage and FileSystem overhead without loosing the advantage of a standardized interface. The drawback though is that the `proc` FileSystem itself is fairly large - so it really only makes sense if it is providing sufficient utility to an embedded system. The question if the `proc` FileSystem overhead pays off is fairly specific to the appliance, but most systems we found had it enabled.

Module size vs. User-space App

One issue related somewhat to the above FileSystem overhead note is the size of user-space applications that would be required to achieve a comparable representation of kernel internal data-structures not using dedicated `proc` files. Such user-space applications not only require storage area on a FileSystem but also the associated libraries must be taken into account. Comparison between a `proc` version of `top` and the usual top program are given later. Generally a kernel-module will be fairly small, most of the `proc` apps we built ended up being smaller than a stripped 'hello-world' using shared-libs ! So the small FileSystem overhead of storing the module can is definitely a clear advantage of this approach. One does need some sort of user-space application to access the files in `proc` but generally these can be considered part of the base FileSystem as they don't need to parse/format data if the content of the `proc` files is already prepared in a user-friendly manner - the user-space can thus be satisfied with `cat` and `echo`.

Portability

When ever you invest time and effort into designing a embedded device the question of portability to other potentially interesting OS/RTOS and hardware platforms arises. It may well be the most significant reason not to go for a `proc` based administrative interface or RT-process control instance as `proc` must be generally considered very non-portable beyond Linux. This concern does not apply though, to the different flavors of realtime enhanced Linux, so in the context of this study the `proc` interface can be considered portable.

/proc functions

This is probably the most significant disadvantage of the `proc` FileSystem utilization for application specific administrative interfaces. It is not portable to other embedded operating systems. The portability over different Linux

supported architectures is very good though, and that is an important issue, anybody that tried to cross-compile 'simple' utilities knows that having cross-platform portability is a serious development advantage.

Bound to kernel release

and it may well even be non-portable between different kernel releases as internal data-structures change quite often.

4.3.6 `/proc/sys` Sysctl Functions via `proc`

The sysctl related functions have type conversions integrated so they provide the safer way of building a proc interface but more restricted. The type conversions are performed in a way that ensures that if incorrect types are passed (i.e. abc to proc_dointvec) then nothing is passed on at all - there is no error or waring though - so checking for invalid/null data is left to the application programmer. Note that the `proc` mirroring of sysctl table entries is a side-effect of sysctl and not vice-versa - mirroring of any sysctl related setups via `proc/sys` can be disabled by passing a NULL string in the procname field. Mode fields are valid for access via `/proc` as well as accessing via sysctl. Naturally the access, particularly of writable files needs to be designed carefully, and should be handled by a security policy not left to the programmers 'intuition' of the importance of a specifical data structure.

4.3.7 Security

A key concern to embedded systems - especially now where every system needs full network access over standard protocols - is security. Two issues out of many should be emphasized here with respect to introducing a `proc` based interface:

- Introducing kernel code is always a potential risk
- Utilizing advanced security mechanism in kernel space can improve security a lot

So security, as usual, depends largely on the know how of the programmers - Linux is not a secure operation system, it is though an operating system that has the potential to be configured and use in a secure manner.

Modifying kernel code

The idea of kernel-space/user-space separation always was that kernel code is validated and safe, but errors in kernel-space often are fatal to the system. On the other hand user-space is considered untrusted, and errors are fatal to the application but not to the system. Introducing Kernel code potentially breaks this trusted-code concept. If a decision is made to introduce kernel code in

a project this requires that a security evaluation is done, which again requires that a security policy is available. The kernel is one flat address space, it is non-preemptive in principal - so deadlock prevention is up to the programmer.

Utilization of kernel capabilities on a file-scope

The last paragraph might suggest that introducing kernel code is in principal a bad idea - the reason why this may not be the case is that the security mechanisms available in the Linux kernel are quite potent but have not really made there way into the FileSystem designs. As `proc` declare there fops on a per file basis, these fops can be designed much more restrictive than a generalized VFS interface, also full utilization of kernel capabilities is possible on a per file basis and this can lead to clearly enhanced security capabilities - as a simple example consider taking away privileges even from the root-user !

4.4 Interfacing to the realtime subsystem

When setting up a real-time task there are a number of issues where using the `proc` FileSystem can help. Notably the starting/stopping RT-threads, reporting status of the RT-system or RT-applications as well as some of the security issues related to managing RT-threads.

4.4.1 Task control via `/proc`

Inserting modules requires root privileges, when setting up a embedded system with RTLinux then one commonly needs some way to launch an RT-thread without giving the operator root privileges. Setting the SETUID bit for insmod is a inacceptably insecure way, as this would allow inserting a trivial module to gain full control of the system. A common method used is to insert the RT-modules at system startup and have the application modules loaded in an inactive state, later a unprivileged user starts the RT-thread by sending a start command via realtime FIFO, but this does requires to give the `/dev/rtf#` write access for unprivileged users, thus also opening some potential problems. An alternative is to use a `/proc` file and protect these files via kernel capabilities if needed. The advantage of the `proc` based solution is that the read/write methods are file specific and not FileSystem specific or tied to the major number of a device with access control restricted to VFS' capabilities, which are generally insufficient, these file specific fops allow very restricted access to kernel space. fops for `proc` files not only map to a very specific read/write method but also have statically, compile time defined, VFS permissions preventing runtime modifications, and allow a very application specific check of passed data.

```
pthread_t thread;
hrtime_t start_nsec;
```

```
static int running=1;
struct proc_dir_entry *proc_th_stat;
```

This RT-thread is launched on insmod (running is initialized to 1) and stops by exiting the `while(running)` loop when running is set to 0 via `/proc/thread_status`, it also allows monitoring the status of this thread by inspecting the `/proc/thread_status` simply by running `cat /proc/thread_status`.

```
void * start_routine(void *arg)
{
   int i=0;
   struct sched_param p;
   hrtime_t elapsed_time,now;
   p . sched_priority = 1;
   pthread_setschedparam (
      pthread_self(),
      SCHED_FIFO,
      &p);

   pthread_make_periodic_np(
      pthread_self(),
      gethrtime(),
      500000000);

   while (running) {
      pthread_wait_np ();
      now = clock_gethrtime(
         CLOCK_REALTIME);
      elapsed_time=now-start_nsec;
      rtl_printf(
         "elapsed_time = %Ld\n",
         (long long)elapsed_time);
      i++;
   }
   return (void *)i;
}
```

One of the nice things about the `proc` files being generated on the fly is that the read method can output the values in a nice user-friendly manner while the write method does not need to bother with any parsing as would be required with a config file.

```
int
get_status(char *page,
   char **start,
   off_t off,
   int count,
   int *eof,
   void *data)
```

```
{
   int size = 0;

   MOD_INC_USE_COUNT;

   size+=sprintf(page+size,
      "Thread State:%d\n",
      (int)running);

   MOD_DEC_USE_COUNT;
   return(size);
}
```

As the proc interface receives character input, one needs to convert input values to the appropriate internal data types - in this example a brute-force atoi is done, which also only takes the first passed character into account. Generally one needs to ensure that ANY write method in proc checks data passed to not open security holes in the kernel.

```
static int
set_status(struct file *file,
   const char *user_buffer,
   unsigned long count,
   void *data)
{
   MOD_INC_USE_COUNT;

   /* brute force atoi */
   running=(int)*user_buffer-'0';

   MOD_DEC_USE_COUNT;
   return count;
}

int init_module(void) {
   int retval;
   start_nsec=clock_gethrtime(
      CLOCK_REALTIME);
   retval = pthread_create(
      &thread,
      NULL,
      start_routine,
      0);
   if(retval){
      printk("pthread create failed\n");
      return -1;
   }
   proc_th_stat=create_proc_entry(
      "thread_status",
```

```
      S_IFREG | S_IWUSR,
      &proc_root);

   /* the file specific operations */
   proc_th_stat->read_proc=get_status;
   proc_th_stat->write_proc=set_status;
   return 0;
}

void cleanup_module(void) {
   void * ret_val;
   pthread_cancel(thread);
   pthread_join(
      thread,
      &ret_val);
   printk("Thread terminated (%d)\n",
      (int)ret_val);
   remove_proc_entry("thread_status",
      &proc_root);
}
```

4.4.2 Exporting RT-process-internals via `/proc`

A critical issue for realtime systems is the ability to monitor status of the system with a minimum overhead. periodically logging to the system logs is one of the possibilities, this is somewhat limited though as the data-volume would become very large and it is often hard to say a-priory what values are going to be relevant for monitoring - so periodic monitoring needs to by adjustable. To make it adjustable a large spectrum of kernel/RT internal values must be reachable with low processing overhead - for this the `proc` and sysctl interface is clearly a most suitable approach. The current `/proc` FileSystem gives you a snap shot of the status of the kernel - but more important for system that need to exhibit fault-tolerance qualities, is the analysis of system tendencies. Roughly this means that the developments of values are more important than the values them selves - with the current concept behind `/proc` there are two possibilities.

- save status locally and periodically compare it to current values
- log status to a remote system and leave complex, and computational intensive, work to a appropriately powerful server system.

with the limited resources of embedded system the first option more or less is not suitable as it would potentially request log/analysis related processing efforts at the same time that the system is in a high load situation due to error handling - thus the data needs to be analyzed as far as possible at low-load situations - this can be best achieved by delegating the data-interpretation to

the systems idle-task, to minimize processing overhead this task is performed in kernel-space and the results are then presented via sysctl or /proc.

Here is an example of making RTLinux internal data available by simply dumping the hrtime variable to user-space via /proc/hrtime - this allows user-space applications direct access to RTLinux internal data structures via open/read/close on proc files or as shown here make it available in a 'formated' way to allow use of cat /proc/hrtime to read the RTLinux internal clock.

```
/* /proc/hrtime "file-descriptor" */
struct proc_dir_entry *proc_hrtime;

/* /proc/hrtime read method - just
 * dump the current time
 * in a human readable manner
 */
int
dump_stuff(char *page,
   char **start,
   off_t off,
   int count,
   int *eof,
   void *data)
{
   int size = 0;
   MOD_INC_USE_COUNT;

   size+=sprintf(page+size,
      "RT-Time:%llu\n",
      (unsigned long long)gethrtime());

   MOD_DEC_USE_COUNT;
   return(size);
}

int
init_module(void)
{
   /* set up a proc file in /proc */
   proc_hrtime = create_proc_entry(
       "hrtime",
        S_IFREG | S_IWUSR,
        &proc_root);

   /* assign the read method of
    * /proc/hrtime to dump the number
    */
   proc_hrtime->read_proc=dump_stuff;
   return 0;
```

```
}

void
cleanup_module(void)
{
   /* remove the proc entry */
   remove_proc_entry("hrtime",
      &proc_root);
}
```

4.4.3 Security Issues

There are some general security issues involved with modules - quite commonly on embedded systems everything is statically compiled into the kernel to eliminate the problem of requiring privileges to load modules at runtime. In cases where this is not possible - and RTLinux is one of them - you need some way to permit usage of dynamically loaded kernel modules in a safe way. For RTLinux a common strategy is to load all RTLinux modules at system startup time (RTLinux core modules + application specific modules), and have the application specific modules in an inactive state (suspended). This way the only thing left to do is to start/stop the RT-threads, which can be done safely via a `proc` interface. GNU/Linux systems since the late 0.99.X releases of the Linux kernel have included the `proc` FileSystem. This FileSystem interface allows inspection of kernel internal data structures as well as manipulation of these data structures. On many embedded systems with tight resource constraints, not only run-time optimization is a requirement but also FileSystem foot-print is a key issue. For such systems utilizing the capabilities of the `proc` FileSystem and the related sysctl functions, to provide kernel related administrative information via `proc` as well as resource-optimized control interfaces can substantially improve embedded systems performance. A further, often ignored aspect, is that the `proc` and sysctl interface allow very precise tuning of access permissions, increasing the security of embedded systems administrative interfaces, and improves diagnostic precision, which is essential for efficient error detection and analysis.

4.4.4 tasklets

A tasklet is a light-wait task, the idea is to have a execution context with limited resources available by default that permits faster context switching. tasklets are scheduled independent of processes/threads and execute at a higher priority that these (both in Linux and RTAI). Tasklets scheduled multiple times are executed ONCE only, the concept was derived from the limitations that the original bottom_half implementations in 2.0.X Linux kernel showed. There prime usage is for DSR routines that are fast and require little resources.

Tasklet functionality available

- Linux kernel tasklets
- RTAI tasklets
- RTAI timers (also called timed tasklets)

The RTAI versions is an independent resource, heavily based on the Linux kernel version, but modified for RTAI's needs.

see section on kernel resources for details on available tasklets

4.4.5 dedicated system calls

The default method for user-space applications to switch to kernel mode is to perform a system call, which is nothing else but a software triggered interrupt managed by the CPU just like a hardware interrupt. On X86 systems Linux uses the `int 0x80` to switch to kernel mode passing the syscall number and possibly arguments. The syscall number is then used to look up the desired function in the syscall table internal to the kernel. This syscall table has 256 entries of which currently only 221 (Linux 2.4.20) are in use (the actual number is kernel version specific, and note also that the syscall interface has been substantially rewritten in the 2.6.X series of kernels).

by inserting a pointer to an application specific function at a free location in the system call table a non-standard system call can be created, permitting a suer-space application to directly call a specific kernel function.

The code-framework for a 'home-brew' system call would look like

```
asmlinkage int sys_test_call(void)
{
   printk("Test System Call called \n");
   return 0;
}
```

Adding the entry point in the kernels system call table (here this is done statically, it could also be done dynamically).

```
   ...
   .long SYMBOL_NAME(sys_test_call)
```

And if this entry is the 222 entry in the system call table then it can be invoked in a standard way by a call to the `syscall` C-library function.

```
syscall(222);
```

The original LXRT implementation and the PSC implementation use dynamically 'registered' system calls (registered in the sense that they modified the system call table at runtime).

Use of this facility makes sense only if a project is implementing a core functionality that will then be used for a larger set of applications, as this allows to provide a clean standard interface. For individual projects this method is too invasive and can't be recommended.

4.5 extended non-standard IPC

In this section we scan some of the available non-standardized IPC mechanisms. With non-standardized we don't mean 'not standard conforming' but IPC concepts where the mechanism it self is not covered by any standard. Some of these non-standard IPC are already covered in other sections.

- networking for socket and RT-net implementations (see part3)
- user-space realtime (see LXRT/PSDD)
- user-space IRQ handlers (see PSC, LXRT and PSDD)

4.5.1 RTLinux/Pro one-way queues

FSMLabs has come to the conclusion that a very light weight lock free communication mechanism, even if fairly restricted, would be able to cover a substantial portion of the communication demands, especially for inter-task communication. For this purpose they developed the so called one-way queues in RTLinux/Pro introduced in Version 1.2. One way queues operate with a lock free mechanism, the internal documentation is currently not available, thus a detailed description is not given.

The API consists of a non-standard (non-POSIX) `sample_enq/sample_deq`. The de-queue operation in user-space is blocking RT-context operates non-blocking (potentially with data-loss on overrun). These queues are conceptually closely related to FIFOs.

```
DEFINE_OWQTYPE - set up a queue
DEFINE_OWQFUNC - assignee enqueue and dequeue functions
NAME_init(); - initialize the one way queue
NAME_deq(); - dequeue a data item from the queue
NAME_enq(); - put data into the queue
NAME_full(); - test fro overflow
NAME_empty(); - test if queue empty
NAME_top(); - dup the top item in the queue
```

The `NAME_` above is a user-defined name passed with the `DEFINE_OWQTYPE` and `DEFINE_OWQFUNC` which is prepended to the one-way queue function calls. The API for one-way queues is the application specific.

(TODO: phase 2 benchmark the one-way queues and compare them with RT-FIFOs and POSIX component message queues).

The actual advantage of this non-standard mechanism is currently not clear to us, unless benchmarks reveal a relevant advantage of this non-standard approach we recommend utilizing standard, POSIX compliant, communication mechanisms.

Chapter 5

User Space Realtime

There has been much debate about the necessity of user-space RT or more precisely memory protection in hard-RT. To state this right at the beginning we don't see this as a critical criteria. Hard-RT applications can hardly follow the concept of untrusted code that is allowed to do anything from dereferencing NULL pointers to overwriting its stack and still should guarantee not to take down the system. The problem here is that memory protection asumes that violations of memory access rules result in termination of the process that caused this violation - this is a resonable strategy for normal user-space applications, but not for hard-RT systems where failure can have a catastrophic effect. Memory protection mechanisms in hard RT only make sense if appropriate exit/recovery strategies can be provided - there is research in this area but still to be concidered an open issue. The principal demand for memory protection for trusted code is not that easy to argue, there are examples of systems that operate without MMU in a flat memory area without any problem, and the discussion would not take place if there was not a price to pay in terms of performance, for having memory protection available. This price of increase context switch times, increased synchronization complexity dues to different virtual address bases and increase in data communication as there no longer is a common global variable realm that is shared, is significant enough to consider user-space RT a second choice only.

At this point memory protection for RT-systems is usefull for development and prorotyping - we recomend that production code should not rely on memory protection unless explicid exit/recovery strategies are included in the design.

A further, very significant, issue that is often overlooked is that even in user-space RT the predominant limitations of RT systems stay in place, one still can't use standard libraries one still can't communicate over non-rt safe methods (blocking I/O) with other user-space applications, and one still does not have the benefits of dynamic resources and non-rt optimizations. So the benefit of user-space RT is fairly limited. Nevertheless it is usable, especially:

- during code development
- for a first step of migrating user-space code to rt-context
- for soft-RT applications
- user-space interrupt handlers

The most significant disadvantage of user-space RT to us seems that it permits a fairly sloppy application design that still will work, and that it does not cleanly split hard-RT from soft-RT and non-RT components of a software system - this split though is at the core of a efficient and maintainable design.

Generally it should be noted that user-space RT will always show a certain overhead compared with kernel-space RT, this processing overhead is in the microseconds range though and tolerable in many cases.

That said to the implementations.

5.1 PSC

During the development of RTLinux/GPL the issue of executing user-space code in hard realtime context arose numorous times in mailing list debates. As one of the first, if not the first, attempts FSMLabs introduced an extension to RTLinux that was based ont the POSIX singal API allowing to execute user-space code as signal handlers for real-time events thus coupling user-space and rt-context.

The definition of PSC varies depending on the publications taken as basis from 'Process Space Control' (Court Dougan, FSMLabs) to Pathetic Signaling Code (Michael Barabanov, FSMLabs). Basically PSC provides user-space interrupt handling capabilities for hard and soft-interrupts only.

The core mechanism of PSC is to provide a dedicated system call, which is dynamically registered by patching the system call table. This system call is used to pass data from user-space to kernel-space. PSC permits registering a handler that is then called from the rt-executive where it is treated as a RT-handler for the assigned interrupt. The handler passed is though execute in the context of the user-space application giving the interrupt handler (refereed to as signal handler) access to the user-application address-space, that is, it has direct access to global variables in the user-space application.

5.1.1 POSIX signals API

PSC uses the POSIX signals API to provide a POSIX compliant interface for user-space handlers to execute on hardware interrupt events.

- POSIX signal functions

  `rtlinux_sigaction`

user-space interrupt handling with PSC

```
#define MOUSE_IRQ 12          /* from cat /proc/interrupts */

void my_handler(int);
struct rtlinux_sigaction sig, oldsig;

/* global variable - shared data */
int scount=0;

int main(void)
{
  /* register handler */
  sig.sa_handler = my_handler;
  sig.sa_flags = RTLINUX_SA_PERIODIC;
```

```
  rtlinux_sigaction( MOUSE_IRQ, & sig, & oldsig );

  /* user-space does work - sleeping here */
  sleep(3);

  /* the handler is reset */
  sig.sa_handler = RTLINUX_SIG_IGN;
  rtlinux_sigaction( MOUSE_IRQ, & sig, & oldsig );

  /* user-space application has access to the data acquired in the
   * interrupt service routine registered via PSC sigaction
   */
  printf("I got %i mouse interrupts\n",scount);
  return 0;
}

void my_handler(int argument)
{
  scount++;
}
```

PSC allows execution of periodic events by binding a handler to the RT timer interrupt this is no different than binding to a external hardware interrupt source.

5.1.2 User-Space ISR

User space ISRs are useful if there is a tight coupling of a user-space application to a hardware interrupt source (telekom devices) or for testing purposes. PSC handlers are limited in what they can do, basically they are limited to what can be done in interrupt context for a kernel space interrupt service routine. PSC user-space ISR is hard-RT when coupled to hardware events, if coupled to timers (or soft-interrupts) then it can not be considered hard-RT as the worst case response time goes up to 10ms (Linux timer interrupt frequency, defined by the HZ system variable).

5.1.3 Limitations of PSC

PSC is basically limited to what can be done in regular interrupt service routines, no blocking operations, no direct library calls, etc.

An implementation limit is that soft-interrupts, that is when PSC uses a signal that is not related to a hardware interrupt but only to a rt-thread pending a soft-interrupt, it will not be executed before the next hardware interrupt occurs. De-facto this means that worst case response time of the handler is equal to

the largest interrupt interval, which is the timer interrupt of Linux if the system is idle - 10 ms on a default X86 system.

Currently, unless a software system really only needs user-space interrupt handling, PSC is not recommended (even though it is possible to build more complex constructs than only user-space ISRs).

5.2 LXRT

LXRT was the first real user-space RT implementation available that rightly carried that predicate. It has been developed over several years now and has a record of projects that successfully applied LXRT for hard and soft-RT demands.

5.2.1 API Concept

The LXRT API always anticipated a fully symmetric API in user-space and kernel-space.

LXRT is provided via a kernel module loaded along with the RTAI system modules that allows user-space LXRT tasks to access all RTAI facilities including scheduling and time management facilities.

One feature of LXRT is that its services are also available to non-root users (TODO: check security mechanisms/risks of unprivileged users using LXRT), which potentially reduces the security issues involved in requiring root-access to manage hard-RT systems.

LXRT also has the limitation, which can be considered an inherent RT related limitation, that no operation in a hard-RT LXRT process may perform any operation that would lead to a kennel mode operation that triggers a task-switch. This includes libraries and access to dynamic resources (notably memory again). It is the responsibility of the programmer to verify that this is not the case if resources other than those defined within LXRT/RTAI are utilized.

5.2.2 Basic concept of LXRT

Basically a non-RT Linux process with scheduling policy SCHED_FIFO is initiated in a rt-safe way (locked memory) and registered with the regular Linux scheduler. With the call to rt_make_hard_real_time() the process is 'stolen' from Linux and from then on managed via a buddy thread that LXRT initiated to provide the timing executive. The process can be returned to Linux by calling rt_make_soft_real_time() within the process. Note that SCHED_FIFO is a RT scheduling policy within Linux aswell, just that it is limited to soft-RT, for this class of Linux processes the recent kernel development has reach substantial improvements.

Using LXRT in soft-RT mode has no advantage over using SCHED_FIFO on regular Linux the advantage it offers in combination with the ability to turn the task over to run under RTAIs control is that a more fine grain split of hard-RT and soft-RT processing can be done without demanding explicit communication (i.e. execute part of a process in hard-RT context and non-critical sections of the same process in soft-RT). The switching process can only happen in non-RT context (in the context of the idle-task 'Linux') so switching is slow and non-deterministic, this needs to be taken into account when designing such tasks.

5.2.3 LXRT

The LXRT subsystem for hard and soft real-time in user-space, allows using all of the RTAI APIs symmetrically in user and kernel space. User-space basically can be considered safe as long as it utilizes the RTAI kernel space API only, if extended resources (libs, shared objects) are to be used it is up to the user to validate the rt-safety of these objects.

5.2.4 New LXRT

New Linux Real Time is a cleanup and extended LXRT again based on a symmetrical API allowing direct interaction between kernel-space and user-space processes (TODO: check mechanisms and benchmark them). It schedules Linux tasks and kernel threads as well as RTAI proper kernel tasks natively. Kernel threads are claimed to run as hard-RT processes (unclear how this is guaranteed as kernel-threads are permitted to do blocking calls). RTAI rt_tasks (hard real-time) also can be instantiated from LXRT modules. User space task/threads can work in any mode, i.e. hard or soft real-time (non-RT is possible but makes little sense if used exclusively) and can switch between modes.

For newer projects NEWLXRT should be used and not LXRT which is to be expected to phase out in the future.

5.2.5 LXRT Modules

LXRT is conceptually modular, as noted above all kernel space API functions are made available to user-space applications, some of these functionalities are packed into modules that are compile time configurable:

- LXRT Real-Time Workshop: interface to RT-Lab
- LXRT FIFOs: allow usage of RT-FIFOS in user-space LXRT modules
- LXRT COMEDI: comedi usage in LXRT user-space modules

5.3 PSDD - Process Space Development Domain

PSDD for RTLinux/Pro, is available under commercial license only. The technological basis currently can't be judged as the source was not available for their first part of the study, from available (marketing) publications the concepts seems closely related to what LXRT is doing.

5.3.1 PSDD API Concept

The API for PSDD is following the POSIX model and targets a symmetric API with respect to the available kernel-space API (TODO: validate POSIX compliance). Some non-POSIX extensions are included, again this is to be seen as a shortcoming of the POSIX standard with respect to hardware related features.

- POSIX time functions

  ```
  rtl_clock_gettime
  rtl_usleep
  rtl_clock_nanosleep
  rtl_nanosleep
  ```

- POSIX file I/O

  ```
  rtl_open
  rtl_close
  rtl_ioctl
  rtl_lseek
  rtl_read
  rtl_write
  ```

- hardware and SMP related functions (non-POSIX)

  ```
  rtl_cpu_exists
  rtl_getcpuid
  rtl_pthread_attr_getcpu_np
  rtl_pthread_attr_setcpu_np
  rtl_pthread_attr_getfp_np
  rtl_pthread_attr_setfp_np
  ```

- POSIX thread attribute functions

  ```
  rtl_pthread_attr_init
  rtl_pthread_attr_destroy
  rtl_pthread_attr_setschedparam
  ```

```
rtl_pthread_attr_getschedparam
rtl_pthread_attr_setstackaddr
rtl_pthread_attr_getstackaddr
rtl_pthread_attr_setstacksize
rtl_pthread_attr_getstacksize
```

- POSIX thread control functions

```
rtl_pthread_create
rtl_pthread_cancel
rtl_pthread_exit
rtl_pthread_join
rtl_pthread_equal
rtl_pthread_kill
rtl_pthread_self
rtl_sched_get_priority_min
rtl_sched_get_priority_max
```

- POSIX semaphores

```
rtl_sem_init
rtl_sem_destroy
rtl_sem_getvalue
rtl_sem_wait
rtl_setm_trywait
rtl_sem_timedwait
rtl_sem_post
```

- syslog interface (non-POSIX)

```
rtl_printf
```

Note that the behavior is equivalent to the functions without the `rtl_` prefix where marked as POSIX functions. All functions are documented in the manual pages of PSDD.

5.3.2 Frame Scheduler

PSDD provides and extended scheduler concept called, eframe-scheduler, this scheduler runs in the context of an application, comparable with user-thread implementations (sometimes refereed to as library scheduler). The frame-scheduler in PSDD operates in the Many-to-one (Mx1) model. Each frame-scheduler provides cyclical scheduler whereby the units used are refereed to as minor cycles, which are a fixed size in the frame-scheduler. Each minor cycle can be

interrupt-driven, or time-driven, it can specify a priority (which can be modified at runtime) and can be allocated to a specific CPU. The scheduling parameters for a task under control of the frame-scheduler are the starting minor-cycle and the frequency in terms of minor cycles, in case of multiple tasks being runnable at the start of a minor cycle the highest priority task is selected.

The basic setup of a frame-scheduler is a

```
while (1) {
   fsched_block();
   do_something();
}
```

5.3.3 controlling the frame-scheduler

The frame scheduler is a user-space RT process launched via a command line interface with the `fsched` command, after launching the frame-scheduler a user-space RT application can be attached to it assigning it the appropriate slot and minor-cycle numbers.

```
fsched create - create a frame-scheduler instance
fsched config - configure minor-cycles and period
fsched attach - atach a user-space RT application
fsched start - start it
```

For monitoring and debugging purposes fsched provides the functions:

```
fsched info - statistic infos about the scheduler and its tasks
fsched debug - debug a frame-scheduler with a specific pid
```

TODO: phase 2 - benchmark the frame-scheduler and evaluate its capabilities especially in the area of automation.

Chapter 6

Performance Issues

In this chapter we will pin-point the software implementation issues that impact hard real-time performance most. Understanding these parts of the actual code is essential to understanding the limits of testing and evaluation. Further more understanding these limits is important for designing analysis methods and specific performance tests to target the demands of a given problem. As this section can't give a complete introduction to the underlaying concepts we will describe the implementations and provide references for further readings.

6.1 scheduling implementations

Obviously one of the functions that will impact hard real-time performance dramatically is the scheduler. Every process (RTAI tasks, or RTLinux threads) can gain access to the CPU by being invoked via a call to the scheduler only, the exception being interrupt handlers, switching to interrupt service routines (interrupt handlers) is done by the hardware without OS intervention.

De-facto every hard real-time operating system will provide a priority based scheduler, or fixed-priority scheduler [21], this is not only the simplest scheduling method from a theoretical stand point it also is one that can be implemented very efficient. Variations of fixed priority scheduling, like RMA, have been developed, but there success was very limited due to the inherent limitations of such algorithms (i.e. RMA is applicable to a set of periodic tasks only with the additional requirement that there execution times and CPU-demands be well defined [22]). Currently there are works known on EDF [23], RMA [**?**], SRP [25] which is implemented with the priority ceiling protocol support in RTLinux. Especially the later priority ceiling and also priority inheritance have been discussed much with the result that there usability for *practical* applications is limited [48]

The limitations of advanced scheduling algorithms is simply due to the fact that the execution time of the `schedule()` function is performance critical, as

an example, the RMA scheduler noted above had to iterate over the entire list of tasks (note RTLinux V1 used a process model, hence tasks), to determine the current period of each task, furthermore it was limited to the assumption that the deadline of each task was the same as its period (that is, there simply is no parameter involved that would describe the tasks deadline in addition to its period). This criticism holds for the EDF implementation as well, which had to extend the scheduler code substantially. And as noted in the section on periodic rt-processes and POSIX compliance **??**. there is a overhead introduced to allow POSIX compliant periodic threads via POSIX-timers, as these also need to be managed by the `schedule()` function [26].
De-facto every hard real-time operating system will provide a priority based scheduler, or fixed-priority scheduler [21], this is not only the simplest scheduling method from a theoretical stand point it also is one that can be implemented very efficient. Variations of fixed priority scheduling, like RMA, have been developed, but there success was very limited due to the inherent limitations of such algorithms (i.e. RMA is applicable to a set of periodic tasks only with the additional requirement that there execution times and cpu-demands be well defined [22]). Currently there are works known on EDF [23], RMA [**?**], SRP [25] which is implemented with the priority ceiling protocol support in RTLinux. Especially the later priority ceiling and also priority inheritance have been discussed much with the result that there usability for *practical* applications is limited [48]

The main consequence of the above notes is that a hard real-time system should at first try to build on a simple priority based scheduler, and only consider more complex solution if this fails. Testing and evaluation of complex schedulers is a non-trivial task and can consume lots of time, even though recently some helpful tools have emerged for hard real-time enhanced Linux variants see section on temporal debugging.

6.1.1 RTLinux/GPL scheduler

The default RTLinux scheduler is a purely priority based scheduler, although there are other schedulers that have been contributed. The scheduler is loaded as a module so basically you can adopt it to your needs and optimize it (don't forget to send the community a patch if you do optimize it ;). This also means that one can code application specific schedulers (generally this means modifying the scheduler core for a specific application, task-number, etc.) without that other services need to be modified.

When loading the scheduler module RTL_SCHED.o the only thread that is registered is non-rt Linux, the 'idle task', and you will not notice much difference. If one performs performance benchmarks on the Linux system in this state one can see that the overhead introduced by the interrupt emulation layer and the scheduling instance beneath Linux cause a performance decrease of less than 1% (taking a PIII 800MHz as reference - this may be different on low end systems). (TODO: benchmark impact of rt-extensions on non-rt performance).

The default RTL_schedule function, will first scan the task-list for armed timers that have expired, if these are found the timer is cleared and RTL_SIGNAL _TIMER is marked in the pending signal mask. In the same run through the task-list the highest priority task is selected. If the hardware would provide an indefinite number of hardware timers there would be no more to do but to find the highest priority task - so we would be done. The X86 hardware only has one programmable timer so the timers need to be maintained as software timers, whats left to be done here is to update the one-shot timer for all tasks whose timer did not yet expire - this must be done at every scheduler invocation as the scheduler itself only has one hardware timer to set, for every thread that could preempt the currently running thread (that is it's priority is higher than the newly selected thread). If no task has a timer armed which may preempt the newly selected thread then the Linux timer interrupt is setup to keep the Linux systems time monotonically increasing.

This is a pseudocode description of the scheduler code RTL_schedule, in rtl_sched.c for the full code refere to `scheduler/rtl_sched.c` in the current RTLinux/GPL repository citertlgpl.

When loading the scheduler module rtl_sched.o the only thread that is registered is non-rt Linux, the 'idle task', and you will not notice much difference. If one performs performance benchmarks on the Linux system in this state one can see that the overhead introduced by the interrupt emulation layer and the scheduling instance beneath Linux cause a performance decrease of less than 1% (taking a PIII 800MHz as reference - this may be different on low end systems). (TODO: benchmark impact of rt-extensions on non-rt performance).

```
rtl_schedule{

   get current time
   set new_task=0
   loop through task list{
      expire all timers, and update one-shot timers
      new_task = highest priority task with pending signals
   }

   newly selected task is not the old task ?{
      switch to new_task
      newly selected task uses fpu ?{
         save fpu registers
      }
   }

   handle the new_tasks pending signals.
```

```
}
```

These scheme is valid for the minimum configuration only, in case that POSIX timers and signals are enabled these also need to be managed. The scheduler also can be optimized at configuration time by disabling the support for floating point usage in case all rt-processes don't need the fpu (this optimization is available in all hard-real-time extension to Linux).

RTLinux/GPL only supports SCHED_FIFO [?], for POSIX compatibility SCHED_RR and SCHED_OTHER are defined, but in fact the policy field of pthread_create is simply ignored (and unless someone comes up with a really good argument for why one needs SCHED_RR in *hard* real-time, this will not change). The POSIX standard definition of SCHED_FIFO is

Threads scheduled under this policy are chosen from a thread list that is ordered by the time its threads have been on the list without being executed; generally, the head of the list is the thread that has been on the list the longest time, and the tail is the thread that has been on the list the shortest time.

Currently SCHED_FIFO in RTLinux does not implement this policy in a standard conform manner (and it is not intended to change this due to performance issues involved) but simply selects between multiple runnable threads in the order they were registered with pthread_create.

The scheduler implementation done in RTLinux performs well even if violating the POSIX standard there is no intention to change this, as it does not provide the order of threads POSIX demands. There should never be a program that depends on the FIFO ordering of the SCHED_FIFO policy, thus the factual ordering should not be a problem. If a program relays on the FIFO order then the program needs rewriting - POSIX is a little inconsistent here as the SCHED_yield() function is conceptually useless if SCHED_FIFO/SCHED_RR define a fixed order.

6.1.2 RTLinux/Pro scheduler

Note that until RTLinux-3.1 there is no difference between the RTLinux/Pro scheduler and the RTLinux/GPL scheduler. Beginning with the RTLinux-3.2-pre releases of RTLinux/GPL the development of the internal scheduling implementation differs.

The RTLinux scheduler as of 2.4.16 kernel provided with dev-kit 1.3 has the following structure:

```
rtl_schedule{
```

```
   get current time
   set new_task=0
   loop through task list{
      expire all timers
      new_task = highest priority task with pending signals
   }

   loop through task list{
      update one-shot timers if task may preempt new_task
   }

   newly selected task is not the old task ?{
      switch to new_task
      newly selected task uses fpu ?{
         save fpu registers
      }
   }

   handle the new_tasks pending signals.
}
```

The current RTLinux/Pro scheduler performs two interactions over the task list. RTLinux/Pro scheduler currently only implements the SCHED_FIFO policy in kernel mode, with the same policy behavior with respect to scheduling order noted above in RTLinux/GPL (again we don't expect this to change as relying on the SCHED_FIFO order is a design error). An extended frame-scheduler is available in the user-space extension PSDD (**??**rame-scheduler) see the section on PSDD for details.

RTLinux/PRO scheduling policy of SCHED_FIFO is not POSIX standard conform (see note above in the RTLinux/GPL scheduler description).

6.1.3 RTAI scheduler

The RTAI scheduler support EDF scheduling and (since RTAI 24.1.6) has support for the SCHED_RR scheduling policy, this scheduling policy can be disabled to improve the scheduling performance of slow systems but it requires editing the scheduler code (by commenting out the macro ALLOW_RR in RTAI_SCHED.c). Currently we see no way how a deterministic system can make much use of SCHED_RR, and there is hardly any theoretical work on this issue, it does make sense for soft-realtime systems, but for such systems the hard real-time extensions based on the dual-kernel model seem unnecessarily expensive. That said, RTAI's scheduler does off course, support a pure priority based scheduler (which is very similar to the original RTLinux scheduler as of version 0.X...)

```
rt_schedule{

  if timer in oneshot mode {
    if SCHED_RR enabled
       update yield time by rr_remaining

    get current time

    if SCHED_RR enabled

       preempt current task and select next one

    loop through task list{
       expire all timers
       new_task = highest priority task with pending signals
    }

    reprogram one-shot timer

  } else in periodic mode {
     if SCHED_RR enabled
     preempt current task and select next one
  }

   newly selected task is not the old task ?{
      switch to new_task
      newly selected task uses fpu ?{
         save fpu registers
      }
   }

   handle the new_tasks pending signals.
}
```

TODO: phase 2 analyze EDF SRP RM deadline-monotonic scheduling for real-life apps, especially the issue of computation quantification and AND/OR tasks (lots of theory and little practical guidance)

The RTAI scheduler has some internal optimizations like checking if reprogramming the timer would not take longer than the time until the timer needs to fire, this is a typical issue of trade-off between scheduling jitter and scheduler optimization. The current implementation calibrates a number of 'tuned' variables that it uses for these heuristic optimizations.

6.2 synchronization

As long as task-sets are independent and preemptible threads schedulability analysis is relatively simple, basically because these two criteria eliminate the problem of priority inversion (A high priority process, ready to execute, blocked on a resource held by a low priority thread). Things change as soon as shared resources and synchronization comes into play, careless application of synchronization objects can lead to unbounded periods of priority inversion or even case a system deadlock/livelock. To eliminate the deadlock issues and to guarantee bounded delays synchronization protocols have been developed, but these are limited in there ability to guarantee acceptable worst-case delays in situations where priority inversion occurs [**?**] - it should be noted here that priority ceiling and priority inheritance make schedulability analysis much more complex, priority ceiling/priority inheritance simply break the assumption of fixed priorities and mean that the process design has a 'built-in' priority inversion problem, this problem needs to be fixed not hidden behind the priority ceiling protocol [27]. One should thought note that there are schedulability theorems available for rate-monotonic scheduling with priority-ceiling, and that these theorems show that the longest duration of blocking in a given task-set can become extremely long - although guaranteed to be bounded.

(TODO: phase 2 design tests and benchmark worst case delays introduced by priority ceiling and priority inheritance protocols)

Even if this may trigger some irritations from the side of the individual providers of realtime extensions to Linux - as this study did not (yet) do any benchmarks of the individual implementations - the performance of RTLinux/Pro, RTLinux/GPL and RTAI/RTHAL as well as RTAI/ADEOS must be considered technologically equivalent and performance wise very similar. This means that the one implementation may be better on one platform or provide a specific feature in a more efficient manner than the other implementation but fundamentally they don differ - which is to day that if you can pin-point where RTAI is better than RTLinux/GPL then it would be a matter of a few days (at most) to improve RTLinux/GPL !

The essential differences between the different implementations with respect to synchronization are the available synchronization objects and how well they fit into schedulability analysis theorems. We consider this an essential part for the proposed continuation of this study, by performing practical tests with the different variants, to validate the applicability of different theorems to the actual implementations.

From the above we derive a TODO-list for the second part of this study (which is not yet under way) to allow for a definite performance judgment of the individual implementations.

- TODO: design tests to measure and quantify performance
- TODO: benchmark the systems with respect to different resource configurations
- TODO: find a common ground for regression testing of the different variants and make them comparable to a well established RTOS (i.e. VxWorks).
- TODO: schedulability analysis (especially the issue of how to integrate different asynchronous event handling strategies, i.e. uninterrupted ISR and DSR).
- TODO: synchronization expenses (especially SMP overhead)

Chapter 7

Resource managment

7.1 Dynamic Memory

One of the key features that is frequently requested for rt-systems and that at the same time bares some quite tricky technological problems and inherent limitations, is dynamic memory in rt-context. Never the less there have been quite a few approaches to this problem, basically one can classify them into two categories.

- hard-limited pool-memory managers
- best-effort dynamically refilled pool-memory managers

Both have their advantages and disadvantages, at present we don't see how a unbound delay can in principal be prevented in the best-effort approaches, which is why hard-limited pool-memory allocators to our understanding are the preferable solution.

As research in this field is on-going, one can expect improvements to emerge and stability and robustness of the dynamic memory allocation mechanisms to increase, it should be noted here though that utilizing dynamic memory in rt-context requires a substantially different understanding of the underlaying allocation mechanisms than is required for writing non-RT user-space applications, this know-how is a requirement and will stay a requirement on the side of the programmer.

7.1.1 Kernel memory management facilities

All hard RT systems are more or less limited with what dynamic resources can be provided in rt-context, the hard RT variants of Linux are no exception to this rule. As a consequence all resources must be allocated and deallocated in non-rt (Linux kernel) context, or in the case of user-space RT allocated at

application startup and then locked (mlock/mlock_all syscall) in main memory to prevent the Linux GPOS from swapping memory to secondary RAM (swap-partitions/swap-files). These limitations hold true for all implementations .

Memory locking guarantees the residence of portions of the address space in physical RAM. It is implementation-dependent whether locking memory guarantees fixed translation between virtual addresses (as seen by the process) and physical addresses, for 32bit Linux based systems the translation is fixed for physical memory below 4GB, for large-memory systems this translation is not fixed, in that case even locking memory will result in a certain overhead on access.

As users don like the idea of static resources, very early in the hard-RT extensions to Linux, different attempts to offer dynamic resources started to evolve. This was partially due to the high resource demands on the system due to static allocation, that would also slow down the GPOS, as well as due to the need for runtime allocation due to language constraints (C++'s constructor methods). The two strategies that evolved are

- allocate large chunk of memory at application initialization and manage it internally
- simply use unsafe calls to the GPOS (de-facto kmalloc with flags set to GFP_ATOMIC)

The first strategy is legitimate and can be used in RT systems although calculating the maximum RAM requirement of an application is a non trivial task in systems that are using dynamic resources. As an example of such an implementation see Victor Yodaikens bget adaptation for RTLinux **??**tl-bget. The net saving of statically allocated RAM is reduced from the sum-total of used memory to the runtime maximum RAM-usage of the application (in theory - practically you can get close to this only). The strategy of internally managing memory resources has found very little practical use as it requires a significant design effort on the side of the application programmer.

The second strategy is unfortunately starting to be accepted in the RT Linux community as it rarely has side-effects on typical desk-top systems - for embedded GNU/Linux systems with low-memory resources this can easily be fatal and relying on such a dynamic memory strategy can not be recommended. Also a commonly ignored limitation of kmalloc is it's 2 to the nth size stepping that results in large overallocation when it comes to larger blocks, a further problem of kmalloc is its hard limit of 128kB (with the additional 'feature' that requesting more than 128kB returns a pointer to 128kB without any error).

As a summary of the above, hard-RT systems require that at least the peak memory usage of the rt-subsystem be locked in physical RAM, general

practice though is to require locking of the sum-total of used RAM at application initialization.

The area of rt-safe dynamic memory may be a valuable are of research in for embedded hard-RT Linux systems - an initiative in this area might be worth considering.

Kernel memory management functions

It is not the intention to give a complete introduction to the memory management subsystem of the Linux kernel but rather to focus on the parts that are relevant for rt-applications.

The process of memory allocation is split into two distinct types in Linux:

- virtual memory - vmalloc
- Fast kernel memory - kmalloc
- Application specific memory slabs - kmem_cache
- Reserved memory at high physical address via kernel boot parameter mem=XYM

vmalloc and friends provide virtual memory of arbitrary size, vmalloced memory is fragmented and may be swapped to secondary memory (swap-space). If memory is allocated with vmalloc due to its size (when ever you need more than 128kB) but must be rt-safe or used in kernel context the pages must be locked to prevent the system from swapping pages to secondary storage. vmalloc is rarely used in kernel drivers.
For details on vmalloc see [52].

kmalloc uses a buddy-system to provide efficient memory usage based on memory being split into power of two size chunks of continuous memory. The requirement of the memory being continuous is why kmalloc is limited to 128kB as it would be very inefficient to maintain larger memory blocks as continuous RAM areas. The buddy system now tries to reunite all freed blocks of a given size to the next largest block, which is why the 2 power n rule is applied. This way applications can get memory from 32bytes to 128kB fairly fast as the chunks are already located and only need to be assigned (as opposed to vmalloc which must grab a set of pages and mark them all in use and then manage them as a continuous address area for the application even if fragmented). The drawback of the buddy system is its inherent limit of 128kB and the fact that requesting any amount of memory will always be rounded to the next power of two, potentially wasting memory if programmers are not aware of this behavior (i.e. requesting 8193 bytes for a structure would result in using up 16kB!). On the

other hand the limit of allocating pages only was seen as unacceptable since many kernel structures are very small, so if used with care, and programmers take the 2 power n rule into account kmalloc is very efficient.

kmalloc has some properties that are of interest for rt-systems, it allows to allocate memory in an atomic way by using kmalloc with the flags set to GDP_ATOMIC, this is rt-safe in the sense that it will never sleep, it is not rt-safe in the sense that it is not guaranteed to return a valid pointer, if no memory was available in the buddy system a NULL pointer is returned. The delayed versions, that are not rt-safe, that is those with flags not equal GFP_ATOMIC/GFP_DMA (TODO: check GFP_DMA), may sleep and request the Linux kernel to free up some memory by i.e. swapping user-application pages to swap-partitions. So the tradeoff is that kmalloc can be fast with a increased risk of returning a NULL pointer or slow with a high probability of returning a usable memory area, in any case kmalloc is not rt-safe as it NEVER guarantees that memory is available , and Linux allows overcoming memory.

The limitation of 2 power n allocation is generally not a big issue, but for protocols that have a fixed size, like IPv4s 1500byte MTU, or applications that need many structures of a specific size, then kmalloc is not the right answer. For such fixed size memory chunks Linux offers the slab cache mechanism. This is memory managed in predefined sizes, similar to what kmalloc provides, just that the sizes are not limited to powers of two but can be set to arbitrary sizes eliminating the memory losses incured by usage of kmalloc (if one allocates 1025 bytes with kmalloc it actually requires 2048bytes of memory). The slab cache mechanism extends Linux kernel memory optimization to a easy to customize facility for kernel space applications.

- kmem_cache_create
- kmem_cache_destroy
- kmem_cache_shrink

and the actual memory allocation/deallocation functions

- kmem_cache_alloc
- kmem_cache_free

As this allows preallocated memory that then can be managed in an application specific way, and will not be shared with Linux kernel functions (unless the programmer uses the same kmem cache in non-rt Linux kernel functions), this memory is rt-safe provided the application never exhausts memory (which there is not cure for in any RT-system).

For memory allocation in rt-context kmem_cache_alloc/kmem_cache_free are a good choice for efficient dynamic memory management
(TODO: SLAB_CTOR_ATOMIC flags description, growing and shrinking cache , SLAB_NO_GROW flags).)

Library Calls

An often posed question is if library facilities are available in hard-RT. The simple answer is no. The fact that one can use some library functions in hard RT context, notably some of the math functions from `libm` statically linked to kernel modules, should not lead to the impression that libraries simply need to be linked as static objects to satisfy rt-restrictions. Basically there always may be *parts* of a library that are rt-safe by accident, but it is in all cases up to the programmer to verify this. This means that even if some examples link `libm` successfully this may change with newer glibc-versions.

It is advisable to take user-space libraries as a guidance for implementing kernel-space libs that are rt-safe, but not use any user-space libs under any circumstances. RTAI has for just this reason begun developing a rt-safe libm, that currently both RTLinux versions (GPL and Pro) lack. Any such library should be designed to be usable in user-space applications aswell, so that testing and validation can be simplified. That said, naturally such a library must be linked as a stoic library object, no dynamic library functions available in rt-context.

The basic method (shown here for the deprecated libm) is shown here to illustrate that it is not anything really RTAI or RTLinux specific - its simply a slightly modified Makefile:

```
...
rt_process.o: rt_process.c
   $(CC) ${INCLUDE} ${CFLAGS} -c -o rt_process_tmp.o rt_process.c
   $(LD) -r -static rt_process_tmp.o -o rt_process.o -L/usr/lib -lm
   rm -f rt_process_tmp.o
```

The code is also nothing unexpected:

```
...
#include <math.h>
...
void thread_code (void *arg){
   double x,f;
   ...
   f=sin(x);
   ...
}
```

Note: Usage of floating point requires the thread to announce this usage as the schedulers optimize by limiting context switch related save/restore operations to the register set actually used. Additionally in RTLinux FPU usage must be configured at compile time.

7.1.2 RTAI memory manager

RTAI provides a memory management subsystem for dynamic memory, at compile time this must be selected to be made available, furthermore it can be selected to use kmalloc (limited to 128kB by default) or vmalloc (fragmented but not limited) should be used, the current default is to use vmalloc.

Basic concept RTAI rt_malloc/rt_free

The core concept is to allocate a large chunk of memory when the memory management subsystem is initialized and then monitor the available memory. If the application layer requested an amount of memory that brings the available memory below the 'low water mark' then a soft-interrupt is used to request further memory from the non-rt, Linux kernel, side. This mechanism is a best effort approach but inherently not bounded. The risk of application working well on development system where memory refills are successful due to systems setup, but failing on a production system where applications may run that were not on the development systems, is fairly high due to the dynamic refill operation that is not rt-safe.

This concept of dynamic memory does not provide a hard upper bounds on the memory resource that can be dynamically requested, thus it is hard to test if a given application will succeed under all system conditions, we recommend to use this strategy with great care and only if the non-rt setup of the production system is well known.

A note on real-time C++ support in RTAI

```
void* operator new(size_t);
void* operator new [](size_t);
void  operator delete(void*);
void  operator delete [](void*);
```

All build on rt_malloc/rt_free, thus the same limitations, risk noted apply to C++ usage in hard real-time.

The use of vmalloc seems like a bad design decision, technically the bigphysarea patch is the preferable way to overcome the kmalloc limitation of 128kB. The use of vmalloc is not recommended. Furthermore a carefully assessment of dynamic memory management facility use in application is a requirement as the memory management system is providing a best-effort but no

guaranteed bounded response time. Application programmers are advised to use the memory management subsystems control functions to inspect the status of the memory subsystem explicitly in application code.

RTAI memory management API

The RTAI memory management API consists of the well known malloc/free type functions renamed by prefixing it with a `rt_`, and additional control functions to check status of the memory management subsystem

```
rt_malloc - allocate memory
rt_free - free memory
rt_mem_init - currently does nothing (?)

display_chunk - displays the allocation details of a chunk.
rt_mem_end - this is exported but seems like it should only be called
                  from cleanup\_module of the memory manager it self
                                (CLEANUP: figure out why its exported)
rt_mmgr_stats - a debug function, it will print the current allocation
                status of the memory management subsystem via ‘printk‘
                                                        (not rt_printk !)
```

Note: there are currently no examples for using rt_malloc / rt_free in the rtai distribution, and the documentation is incomplete (see `rt_mem_mgr/README`).

7.1.3 RTLinux/GPL DIDMA (Experimental)

Doubly Indexed Dynamic Memory Allocator - currently this module is external to RTLinux as it is still considered experimental (it is expected to move into the official release soon though).

Due to developers preference, this package is also named TLSF, Two Level Segregated Fit, but this does not impact on the concept. The final name of this package seems not yet to be settled...

RTLinux does not provide any kind of memory management support, neither virtual memory (by mean of the processor MMU page table or memory segments) nor simple memory allocation, as the one provided by the standard 'C' library. RTLinux applications must preallocate all the required memory in the mandatory init_module() function before any rt-threads are created. Once the RTLinux threads are created, the only rt-safe memory that can be used is the treads stack (kmalloc can be invoked but is inherently not rt-safe).

The main reason for not providing wrappers to the Linux kernel memory management functions in RTLinux is the that both virtual memory and dynamic storage allocator algorithms introduce a unbounded delay in the rt-system, making the system response unpredictable and inherently non-RT.

The DIDMA allocator implemented a new algorithm, designed with the aim of guaranteeing bounded response time. The new algorithm is called DIDMA (Doubly Indexed Dynamic Memory Allocator). It is based on the use of a pure segregated strategy. See [**?**] for details.

Basic concept of DIDMA

DIDMA uses the 'bigphysarea' patch to the Linux kernel to overcome the limitations of the kmalloc kernel function (limited to 128kB of continuous memory in mainstream Linux)

When the DIDMA.o kernel module is loaded , it reserves a big block of memory using 'kmalloc' kernel function, this memory is persistent (can't be swapped). This module export the core memory management functions rtl_malloc, rtl_free, rtl_realloc, etc... for rt-threads. On removal of DIDMA.o the kmalloced memory chunk is kfreed again.

(TODO: phase 2 benchmark bounds for allocation, validate concept of DIDMA)

DIDMA API

The API is similar to the known malloc,free allocation functions, prefixed with rtl_ (#include <rtl_malloc.h>).

```
rtl_malloc - returns a pointer to the assigned area if any available
rtl_free - returns the area to the linked list of objects
rtl_realoc - realoc equivalent
rtl_calloc - calloc equivalent
```

DIDMA kernel module also implements some debug related non-standard API for dumping and inspecting of memory areas allocated with the rtl_ memory functions.

This extension to RTLinux can currently not be recommended as it is still more or less untested, but it may well be stable within a short term (targeting the next RTLinux/GPL release).

Chapter 8

Hardware access - Driver Issues

Drivers for realtime enhanced Linux have always been a considerable problem, on the one side due to the vast number of different products and the unwillingness of many vendors to provide adequate infos, on the other side due to the often non-standard methods of configuration and accessing. The only note-worthy project for realtime Linux drivers is currently the comedi package [57] that is available for RTAI and RTLinux as well as for non-RT mainstream Linux. Some additional projects in the area of real-time communication for distributed systems are also available (see the section on real-time networking) - in here we are more concerned with device drivers for data aquisition and actuator control.

Aside from this package it should be noted here that the Linux kernel has all necessary provisions available to allow for easy configuration of PCI and ISA (aswell as other bus subsystems) devices and bus specific resource initialization, basically this reduces the task of a driver writer to the device specific I/O functions and the associated data item management (synchronization and buffering).

In the following section we will address the issues of synchronisation, data management and security. A short, and non-exhaustive, section on platform specifics is appended.

8.1 synchronization

In normal Linux derives protecting critical data objects is fairly simple as all that needs to be done is to guarantee atomicity (that is excluding performance issues for now) in realtiem context this is not quite as simple as 'brute-force' synchronization easily results in priority inversion or even deadlock problems and long code sequences that run protected by disabled interrupts increase the scheduling jitter and interrupt response latency in an inadequate way for a realtime system.

Synchronization is one of the prime sources of subtle problems with realtiem drivers as there are a number of factors that distinguish realtime drivers from non-realtime drivers.

- fast-path slow-path optimization fails ??erminology
- DSR strategies only possible with limitations in respect to schedulability [28]
- ISR context 'random'(thus limited with respect to synchronization)
- fine grain synchronization required
- hardware access may influence realtiem behavior (DMA, burst PCI, slow ISA)

The basic strategies are available for asynchronous event handling in realtime systems [21]:

- allow the ISR to interrupt any periodic task and run to completion
- set up a DSR and execute the interrupt service in a defined context.
- force the ISR to run with a priority lower than any periodic task.

This three solutions have clear implications on the hard-realtime behavior of the system, solution one is tolerable if, and only if, the ISR is very short, basically if the expense of invoking the scheduler is higher than strategy one is fine, this is the case if a driver needs to do no more than do I/O management and update some management related data structures but not actually copy data or process-it.

Strategy 2 is the preferred way to go as it allows analysis of the system as the asynchronous event becomes a thread that can be treated as a periodic event (a thread 'polling' the interrupt, the interrupt service is reduced to minimum hardware handling (ACK interrupt, manage peripheral registers as needed) and marking the rt-process for execution.

The third option is only applicable to systems that are doing synchronous processing only ('pure signal generation') and have no hard-realtime demands on peripherals, generally this is not the case and making interrupt lower priority than periodic rt-processes is not an option (it leads to non-deterministic latencies on interrupts). In cases where this strategy is considered the way to do it is to let Linux (the non-rt GPOS) manage these peripherals.

As noted above ISRs run in an undefined context, the UNIX tradition is not to set up an interrupt specific context as this would require two context switches on every hardware interrupt, but to simply execute the ISR in the

context present on occurrence. This basically means that one can not rely on availability of any context specific data items (local variables), but this is not a real problem as ISRs are executing in kernel space and thus have access to global kernel variables. It should be noted though that access to these must be synchronized in an interrupt safe way (non-blocking)

The issue of fine grain synchronization arises as soon as larger data items need to be copied/modified in interrupt context, generally this is a bad thing to do and the preferred strategy is to copy data to a 'safe' location and then manage modification with fine grain locking as to minimize times where locks are held (potential priority inversion problems).

8.1.1 buffering

As soon as larger data amounts need to be passed between peripherals and rt-processes global variables are insufficient. The issue of buffering of data appears as soon as DSR strategies are involved which is the common case. Buffering related problems are basically:

- restricted (lack) of dynamic resources
- large data blocks copied uninterrupted cause high system scheduling jitter
- making data management routines (data copy/compression) reentrant is very complex
- performance issues with copying data
- hardware effects of buffering (cache flushing, context switch when user-space is involved)

The problems related to large data blocks being transfered at once that block the system during this copy operation (especially with DMA) have no software solution , they must be solved at the design level - it must be clear to application designers that there are time-limitations to data transfer from/to peripherals that need to be taken into account.

The restricted dynamic resource availability, notably dynamic memory, mandates that one allocate all required memory at application start, strategies to reduce this amount are allocation of memory pools with internal management, but this requires that the maximum amount of memory that will ever be used at one time is know (which is not easy to figure out). Unless this maximum can be cleanly calculated, usage of dynamic resources (bget port to RTLinux or rtai_kmalloc/rtai_kfree in RTAI) are not recommended - to say it clearly - runtime testing of applications with dynamic resources is insufficient, validation of dynamic memory must be possible from the design.

The overhead of buffer copying can be reduced by using zero-copy interfaces (also known as buffer swinging) there are some IPC implementations that utilize

this strategy (message queues in RTAI (CLEANUP: check code on this) one-way queues in RTLinux/Pro).

8.1.2 security

All though security naturally is not a problem specific to realtime drivers, it is noted here as there are limitations inherent to realtime systems with respect to drivers.

- limitations in sanity checks at runtime for parameters
- limited error processing capabilities (processing overhead, limitations due to exit strategies)
- error correction
- logging and monitoring
- data integrity (all drivers are in kernel-space even if processing can be moved to user-context)

Chapter 9

CPU selection Guidelines

This is to be considered preliminary and intentionally was to become a 'CPU selection Guide'. As this study did not (yet) do any hardware tests, this section is notoriously incomplete.

9.0.3 Introduction

Many CPUs are well suited for general purpose OS usage even though they may have some inefficiencies or even hardware bugs fixed in software. As a regular user with response times in the 10s of microsecond... one does not notice these issues unless one uses such a system for real-time constraint applications, like a simple audio-player.

The problem is that real-time is influenced by the entirety of the system and not by a single component that can be easily isolated, we will point out some of the typical components that cause problems, and in the section 'Preliminary Testing'we provide some guidance on validating a hardware platform for use with one of the hard real-time extensions to Linux. It should be noted that there is no relation between the expenses of a system and its suitability for hard real-time, we have had high-end server systems that were unusable and off-the-shelf noname-PCs that showed excellent performance !

9.0.4 RT related hardware issues

As said above the entire system setup influences the real-time characteristics of a system, there is no way around actually testing a system. The hardware components noted here are some of the subsystems know to cause problems, keeping away from these will increase the probability of a system being usable for real-time

Cache

Generally modern CPUs provide data and instruction caches, these optimize average performance, by providing a small, the cache, area that is faster than RAM, to store recently accessed data in. These caches generally are split into instruction cache (ICACHE) an data cache (DCACHE). The draw-back for real-time is that these caches are much smaller than the RAM installed and that they need to be flushed when the page-range being accessed in RAM changes, as data-integrity can't be guaranteed during such a flush operation, the CPU de-facto is stalled during a cache flush. Thus larger caches can cause substantial delays in a system (reported to be in the range of 10s of microseconds for large 512kB caches on a Pentium III system, TODO: benchmark a few systems to quantify this more precisely)).

We recommend considering/testing system with small caches, for real-time systems, first.

Buses

This section is simple - generally:

- Keep away from ISA buses if possible
- Keep hard real-time devices off PCMCIA buses
- USB is inherently non-real-time
- don't share interrupts (see above)

Peripherals

The bad news is hardware needs to be designed for use in hard real-time systems, The good news is, many simple devices are well suited for hard real-time. Again what we noted above holds true, the suitability of hardware components for hard real-time systems does not correlate with expenses ! Very inexpensive, but simple, peripherals generally are more deterministic then highly, hardware, optimized devices.

That said - again there is no way around testing a peripheral component, and testing MUST be done in the integrated system to allow definitive judgment. Testing peripherals isolated from the integrated system will result in incorrect (optimistic) results.

9.1 Interrupts

Asynchronous hardware events strongly influence the behavior of a RTOS, this influence is so dominant that the interrupt response times and the interrupt induced scheduling jitter is a typically measured value for hard-realtime systems.

As the interrupt response, the ISR, is triggered without CPU intervention, hardware interrupts are not directly under the control of the RTOS, which is why the strategy for interrupt management influences the quality of an RTOS very strongly. Aside from this interrupt management strategy there are some hardware related issues that the RTOS can't influence, this problems are hardware related and need to be taken into account when selecting the CPU and the closely related motherboard setup.

Factors that influence interrupt behavior are

- sharing of interrupts
- interrupt sources that are outside of the control of the OS
- interrupt controller hardware

9.1.1 Shared Interrupts

With very few exceptions (LNET IEEE-1394 driver) shared interrupts are not supported by any of the currently available hard-realtime Linux extensions, for the preemptive-kernel variants (soft-realtime Linux) this is not true, these generally support sharing interrupts over different devices. It should be noted though that sharing interrupts for realtime devices is a design error in almost all cases, especially for PCI systems this is not necessary for normal PC systems as the dasychained PC-interrupt routing can always be assigned without sharing interrupts [45].

We do not recommend sharing interrupts, and generally it is not necessary, if for some reasons this sharing must be done then one can build a driver following this framework - it should be noted though that this introduces additional jitter in the system due to increase in the minimum interrupt handling that needs to be done.

```
pthread_t dsr_thread;
static struct sigaction oldact;
static int shirq=SHIRQ_NR;

void rt_irq_handler(int sig) {
  int ret;
  ret=check_rt_hardware();
  if(ret){
     handle_rt_hardware();
     pthread_kill(dsr_thread,RTL_SIGNAL_WAKEUP);
  }

  /* pass on the interrupt to Linux */
  pthread_kill(pthread_linux(), RTL_LINUX_MIN_SIGNAL+shirq);
```

```
}

void *dsr_thread (void *param)
{
  ...
  /* do data processing outside of the ISR */
  while (1) {

     process_data();

     pthread_kill(dsr_thread,RTL_SIGNAL_SUSPEND);
     pthread_yield;
     pthread_destcancel;
  }
  return 0;
}

int init_module (void) {
  struct sigaction act;

  act.sa_handler = rt_irq_handler;
  act.sa_flags = SA_FOCUS;

  rtl_hard_disable_irq(shirq)

  /* set up interrupt handler */
  if (sigaction (RTL_SIGIRQMIN + shirq, &act, &oldact)) {
    rtl_hard_enable_irq(shirq);
    return -EAGAIN;
  }

  rtl_hard_enable_irq(shirq);
  return 0;
}

void cleanup_module (void) {
  /* reset to old handler */
  sigaction (RTL_SIGIRQMIN + shirq, &oldact, NULL);
}
```

Note that this framework holds with RTAI aswell, using the equivalent non-POSIX RTAI functions, This also is an example of the limitation of POSIX with respect to hardware related functions, POSIX provides no standardized facilities to manage specific interrupts which is needed here during hardware initialization.

9.1.2 CPM

The MPC8XXs Communication Processor is a multi-interrupt capable device that appears as a single interrupt source to the 603 core CPU, this construction is in some ways very effective but it has a serious draw-back for hard real-time systems, the CPM interrupt is potentially shared by 29 (!) event sources, resulting in very high latency. Furthermore the CPM has a fixed priority on the interrupt-sources that can't be modified at will, thus limiting the usability of CPM based devices for hard realtime. For a peripheral device that has hard realtime constraints the use of CPM interrupts is deprecated, even if this may seem to eliminate the advantage of the MPC8XX system almost entirely, this rule allows to build valuable hard realtime systems as the reverse implication of this is that many non-realtime services can be delegated to the CPM, thus reducing the load on the core CPU. If the CPM is assigned to non-hard-realtime events (communication, networking, etc.) and the limited free IRQ-pins to the 603 core are utilized for hard-realtime demands the MPC8XX can be a valuable hard-realtime platform. In any case it should though be noted that a very careful design of the interrupt layout is necessary in the case of CPM usage.

9.1.3 SMIs

System Management Interrupts (SMI) are inherently evil things for an RTOS, the SMI is a hardware feature of a processor and there is basically no way to get around it (disable it or emulate it). SMIs have been used to emulate peripheral hardware (sound-blaster compatible sound subsystem on GEODE processors) or to fix hardware bugs (APIC fix in the MediaGX). In some cases the SMI extensions can be disabled and that resolves the problem, in others where it is fixing hardware bugs this disabling is not possible and thus such a CPU should simply be excluded from any selection process that targets hard-realtime demands.

9.1.4 8254/APIC

Although the 8254 timer chip is more than out-dated it is still being maintained for compatibility reasons on many X86 platforms, and still used on a number of SBCs. Generally access times to the 8254 are lousy and need to be explicitly benchmarked to ensure the time-stamp resolution of the system is sufficient (the 8254 timer resolution is 838 nano-seconds which is in principal sufficient for most systems) especially because accessing the external timer chip can be very slow and can be influenced by other system activity, notably DMA transfers.

If possible APIC based X86 systems should be preferred over systems with 8254 chip-set. RTLinux/GPL and RTLinux/Pro do not support direct manipulation setting of timer behavior, it is done via the API functions implicitly, on

the contrary RTAI provides timer management functions to change hardware timer related settings:

- one-shot-mode - `rt_set_oneshot_mode`
- periodic mode - `rt_set_periodic_mode`
- start 8254 - `start_rt_timer`
- stop 8254 - `stop_rt_timer`
- start APIC - `start_rt_apic_timer`
- stop APIC - `stop_rt_apic_timer`

This allows optimization for specific task-sets (rate monotonic and common time-base tasks).

9.2 Platform specifics

This section is preliminary as no test were performed during this first part of the study, the information quality is thus limited to general statements. Intentionally this section should become an independent *cpu selection guide* later.

9.2.1 ia32 Platforms

X86 platforms should be split into three categories of systems:

- embedded uniprocessor systems
- desk-top class systems - especially for development and test-systems
- server class - notably SMP - systems for high end RT-systems

We propose to design and implement a reasonably standardized test suite based on the real-stone and trigraph tests to classify X86 CPUs for use with real-time enhanced Linux systems (TODO: phase 2)

- AMD
  General performance of AMD processors has shown to be above those of Intel equivalent system when viewing pure CPU performance, notably the embedded processors AMD SC410 and SC520 show outstanding performance if compared with comparable cases (i486/i586/
  PentiumI) due to on-chip timers and hardware design details. Noteworthy in this context is that the SC4XX and SC5XX processors can be operated fanless. For the high-end systems the clearest advantage of AMD processors especially the DURON class processors has shown to be its

small cache which makes memory access slower in average but reduces the worst case incurred by a flush_all TLB flush. Information on current CPUs is not yet available in a reliable way (notably AMD-XP).

- Intel
  Although dominant in the mass market, Intel CPUs for RT applications have not been as successful at least when it comes to RTAI/RTLinux applications due to some of there hardware features, notably the large caches on PIII class systems, and the lousy performance of the small caches on P4 Celeron systems (tests on P4 Celerons are preliminary though - only very few sources of info and no precise benchmarks). The class of mobile CPUs has shown problems with RTLinux and RTAI due to the inability to disable power-management effects (claims are that on these CPUs even with disable power management some power saving strategies are still active). Intel systems clearly dominate when it comes to SMP systems, notably dual-Celeron systems show good performance, and in the high-end range of the Xeon multiprocessors successfully application with RTLinux have been reported , although no reliable numbers are available, especially with respect to multi-threading on the Xeon system.

- Syslogic Even though syslogic is a very small company that has a limited portfolio of embedded systems, there NetIP series of embedded processor boards has shown good performance, this is in part due to the CPU selected (ST586) but seems more related to the system integrating quality. Reports on the NetIPC 1A,2A and 2H are known and showed good overall performance. A noteworthy advantage of the Syslogic systems is their fanless operation.

- VIA Especially in the area of fanless devices the VIA-EDEN (CIII) is one of the highest performing CPUs around, generally the latest VIA CIII based SBCs show excellent performance numbers, which seem mainly due to the system integration quality (all chips on the VIA produces boards are from VIA). Generally the VIA CII and CIII processors have had a lot of positive reports (not that this does NOT include earlier Cyrix processors). As of writing the VIA CIII based SBCs also provide the best cost/performance ration in the X86 embedded market.

9.2.2 PowerPC Platforms

- Motorola

- IBM

9.2.3 Platforms known to cause problems

- MediaGX
  There have been a number of reports on problems including extremely high latency (in the hundreds of milliseconds) and bad overall performance

- GEODE
  Many reports of problems, high jitter and high latency, aswell as poor floating point performance

- Large L2 Cache
  Generally systems with large L2 caches can show large jitter, the L2 cache shout be selected as small as necessary to provide the required performance, generally the rule of desk-top and server systems, that larger caches improve performance, is false in RT systems.

- Notes on SMP systems Unfortunately one can not say anything about an SMP system based on numbers obtained from the same CPU in a UP system, in SMP systems the motherboard (or more generally, the system integration) is the key issue for performance. We recommend extensively benchmarking a SMP system before selecting it for a project.

- Notes on 'mobile CPUs' (laptops) Keep away from the mobile CPUs if possible. If a mobile CPU MUST be selected for a project then we recommend implementing a strong monitoring system or operating the RT-system with one of the tracer packages, excessive jitter in laptop systems has been reported to show stochastic behavior and to be very hard to reproduce, thus error analysis requires to have temporal data available. The mobile CPUs are simply not designed for RT appliances and can't be recommended for real-time enhanced Linux systems.

We are well aware of the fact that the guidance provided in this section is insufficient, as of writing the available data is scarce, and more problematic, incomplete, a systematic analysis of these issues is recommended.

Chapter 10

Debugging

In GNU/Linux systems GDB can be called THE standard debugger, with a large set of external modules and patches allowing platform and OS specific extensions. Aside from these extensions the design of GDB includes the concept of remote debugging which also is utilized for debugging of kernel core code and realtime extensions that operate beneath the GNU/Linux OS.

GDB is a classical source code debugger, the key mechanism (with some hardware variants) is to utilize trap gates to allow single steeping of execution. Naturally under these conditions realtime constraints can not be fulfilled. This is at the root of the core problem of debugging of realtime tasks. Debugging must be split into two distinct levels

- source code debugging in non-realtime (serial execution debugging)
- temporal debugging at runtime (temporal debugging)

10.1 Code debuging

The classical code debugging with a source level debugger like GDB is not always sufficient for low level debugging of the underlaying Linux kernel - although not related to realtime extensions using the dual kernel strategy, these extensions are essential for debugging of the core GPOS:

gdb

GDB (current production version 5.3) is a stable source level debugger that support anything that can run Linux. It is well tested and well maintained and has an active community for user-questions and for developers.

- Homepage: `http://sources.redhat.com/gdb/`
- Download: `tp://sources.redhat.com/pub/gdb/releases/`

- Platforms: X86,PPC, anything that supports Linux
- Comment: Well established 'standard' Linux debugger, well documented, well supported - no reason to use a different debugger for code-debugging under Linux. Support remote debugging of multiple targets and the community has developed a number of extensions some of which are RT and embedded Linux related (see below).

kgdb

The Linux kernel debugger - RTAI relies on KGDB for kernel level debugging, this is possible in RTLinux as well but not required. The current version of a suitable kernel GDBstubs package can be obtained from kgdb.sourceforge.com. The debugger runs in client server mode with `/sbin/ gdbstart` launched on the target board (application is compiled along with the patched kernel image for the target system). On the user-front end simply run GDP as remote debugger via serial line `target remote /dev/ttyS0`, the sources for the target must be on the user-host in an unstripped version. development:/usr/src/linux/arch/i386/ kernel

- Homepage: `http://kgdb.sourceforge.com`
- Download: `http://kgdb.sourceforge.net/downloads.html`
- Platforms: X86 others ?? (its always a bit behind on other arches and normally not up to date with the latest kernel)
- Comment: KGDB is a patch to the Linux kernel, debugging can be done via serial lines (CLEANUP: ethernet supported in all arches ??), documentation can be found in Documentation/i386/gdb-serial.txt (in the Linux kernel documentation tree after applying the patch).

bdm4gdb

PPC specific background debug module for GDP (operates via JTAG port on mpc8XX, mpc82XX not supported (CLEANUP: check that this has not (yet) changed).

ksymoops

Ksymoops is more of a error report tool than a debug tool although it is very helpful for debugging of kernel crashes (so called oops events), ksymoops allows to map the stack backtrace to the involved functions via the kernels symbol-file (System.map) - Oopses piped through ksymoops are the preferable way of reporting any kernel bugs to the Linux kernel developers. Generally responses to oopses that are posted in ksymoops preprocessed form is quick (minutest to

hours). Posting the oops output is more or less useless to the developers as the addresses are kernel specific. For embedded systems the 2.6.X series of kernels has a built in ksymoops allowing human-readable-oops'ing.

- Homepage: none (?) contact: `ksymoops@ocs.com.au`
- Download: `ftp://ftp.kernel.org/pub/linux/utils/kernel/ksymoops`
- Platforms: all platforms that support Linux
- Comment: Every embedded engineer working on Linux should know how to use ksymoops...

A patch against sysklogd 1-3-31 `patch-sysklogd-1-3-31-ksymoops-1.gz` to preserve information required for ksymoops is available at the ksymoops download site. This patch has been accepted by the sysklogd maintainer and should appear in the near future (possibly next sysklogd release). This is essential for post-mortem analysis of systems that are not monitored all the time but have syslogd running.

rtl_debugger

The RTLinux debugger implements the GDP stubs specifically for RTLinux, this allows catching exceptions from RTLinux threads safely and allows to build the debugger around the demands of RTLinux. Conversely the KGDB implementation of the GDP-stubs targets the Linux kernel, as the hard real-time extensions to Linux are operating in kernel space this automatically permits a certain level of debugging as part of the Linux kernel.

The rtl_debugger was originally implemented by Michael Barabanov for the RTLinux-2.2 release, and has since been maintained as integrated part of RTLinux/GPL, it is to be expected that this will not change in the near future as the rtl_debugger allows debugging very deeply into the RTLinux source (by loading the appropriate module symbol tables). Conceptually the rtl_debugger is a remote debugger - that is the GDP-stubs provide the data via `/dev/rtf10`, just like KGDB does via serial line, this allows to use rtl_debuger on the local host via rt-FIFO or on a remote system by means of netcat. A further interesting feature of rtl_debug is its ability to connect to a faulting task after the fault occurs, this allows rtl_debug to be loaded on production systems for high-level post-mortem analysis. Current versions available with RTLinux-3.2-pre3 support GDP up to version 5.3.

RTLinux/Pro also support the rtl_debugger, beginning with RTLinux-3.1 the development and maintenance of rtl_debug is basically independent in the two trees, but it is expected (and anticipated by FSMLabs) to keep it as compatible to RTLinux/GPL as possible.

An advantage of the independent implementation of the GDP-stubs is that rtl_debug require no patching of the Linux kernel or the RTLinux sources specifically for use with the debugger, its ability to provide local and remote debugging sessions due to the RTLinux specific communication mechanism implemented. Further more rtl_debug uses the syslog interface (via rtl_printf) to notify the Linux side of the fault occurrence and this with the option of launching GDP after the fault actually occurred is a clear advantage over typical source code debuggers that require debug sessions to launch an application in the debugger.

It should be noted though that one debug-mode is entered no timing constraints can be met any longer.

graphical front-ends to GDP

The debuggers discussed up to here all provide a text-mode user interface, but the data format is in principal independent of the representation, so a number of graphical front-ends have been developed, the list here may not be complete, we chose to only list graphic front ends that we found report of successful use with one of the hard real-time variants in discussion.

- xgdb
  - Homepage:
  - Download:
  - Platforms: all that support gcc (more or less anything with 32bit)
  - Comment: Well tested, widely in use, reported to be ugly.
- ddd
  - Homepage:
  - Download:
  - Platforms: X86, PPC (others ?)
- Insight:
  - Homepage: `http://sources.redhat.com/insight/`
  - Download: `ftp://ftp.gwdg.de/pub/linux/suse/ftp.suse.com/suse/ i386/8.2/suse/i586/insight-5.2.1-133.i586.rpm`
  - Platforms: X86, cross-debugging of PowerPC, Hitachi SH reported.
  - Comment: Written in Tcl/Tk for GDB - Little use in RTLinux/RTAI reported - but it has been reported to be used (TODO: verify/validate insight for RTAI/RTLinux debugging)

10.1.1 Non-rt kernel

In a patched realtime enhanced system it is not only of interest what the RT-kernel is doing but it is also interesting to know what influence the RT-subsystem has on the non-rt kernel. For this purpose there are a number of profiling tools around.

- kernprof (SGI open-source kernel profiler project)
- gprof+uml (TODO: check for recent kernels)

10.2 Temporal debuging

To guarantee the proper operation of realtime systems it is not sufficient to guarantee alone that code is executed in the intended sequence, as in non-rt code, but also the temporal behavior must follow the specifications. Validating the temporal behavior and it satisfying the temporal specification is fairly complex due to:

- the entire environment influencing the temporal behavior (software and hardware)
- unpredictability of asynchronous events
- complexity of synchronization in multi-threaded applications
- recording overhead
- time stamp limitations of the system

The first two issues noted are coupled and are the most critical as it implies that verification of temporal specifications in principal can NOT be done by mere test runs unless the entire environment is sufficiently specified to allow prediction of all external events (external in the sense of not being a direct part of the rt-executive). Preferably some for of formal validation is necessary, although it has been shown that ‘staking‘ worst cases, as a typical conservative approach might suggest, results in unusable ‘horror‘ numbers.

Some of the factors that have a strong influence on temporal behavior that can’t be recorded directly are:

- hardware-cache(s)
- hardware error correction mechanisms
- bus-topology (DMA,shared interrupts, cascaded-buses)
- external interrupt sources (desk-top systems typically have about ten interrupt sources that are hard to predict)

The third issue noted above, synchronization complexity, is especially aggravated by concepts like priority inheritance and priority ceiling protocols, as well as specifics of the scheduling policy (FIFO,round-robin,EDF, etc). Validating temporal behavior of a multi-threaded system relying heavily on thread synchronization mechanisms can not be done without internal knowledge of the core RTOS system, this is a limitation on the side of the developers not the system and the only way out is that developers of RTOS systems need RTOS knowledge to perform temporal system validation. Tools that do help are tracers (see below) but they don't eliminate the demand for RT specific know-how on the side of the developer.

The last two points noted above, recording overhead and time stamp limitations of the system, are also RT-system inherent. Recording the sequential instruction path of a system requires a certain processing overhead for the recording instructions and the memory access for trace-data storage. Fortunately tracers have been developed that have a sufficiently small overhead (¡5

TRACE TOOLS

As the problems of temporal debugging are inherent in RTOS, the development of tools to cope with these problems evolved fairly early in the various flavors of hard-realtime Linux.

RTLinux tracer The RTLinux tracer, implemented by Michael Barabanov, was the first approach to temporal debuging in hard real-time Linux extensions. The basic concept is to have shared memory areas that provide a predefined number of event-buffers, these event-buffers are continuously written to at critical points in the core OS and the application code. The application then can trigger a trace by calling `RTL_TRACE_FINALIZE`, which then causes the tracer to switch to the next free event-buffer and continue recording. This method allows to backtrace temporal dependancies starting at the event of interest and thus analize the hot-spots in the code. Naturally this can't happen without a cirtain overhead for the recording process, but this overhead is in the range of 1%.

The RTLinux tracer is integrated in the main RTLinux/GPL development tree and is also part of the comercial RTLinux/Pro distribution.

Linux Trace Tool kit, LTT LTT was originally developed for kernel development of the mainstream Linux kernel and is still maintained for this purpose, it is a good tool to move into the internals of the Linux kernel.

http://www.opersys.com/LTT/downloads.html

POSIX tracer The POSIX tracer is a kernel module that performs a similar event trace as the rtl_tracer module. Debugging of complex rt-applications

requires a method of analyzing the actual flow of control in the temporal dimension, the IEEE has incorporated tracing to the facilities defined by the POSIX standard, this POSIX Trace standard .

The POSIX tracer developed by the OCERA group, has some analytical interfaces (see the section `kiwi` below) and runtime interfaces for fault tolerance (see `ftappmon` below). It allows a temporal analysis of the individual rt-threads as well as overall system performance monitoring based on logging critical events.

http://www.ocera.org/download/components/WP5/ptrace-1.0-1.html

LTT for RTAI RTAI support in LTT is available as of LTT version 0.9.3, tested on X86 and PPC platforms. Intentionally it is used for monitoring and analyzing of RTAI based systems behavior , this is thus not only a temporal debugging tool but also a code validation tool intended to help understand real-time embedded systems. It is a valuable tool for technicians starting into the hard real-time enhanced Linux world.

LTT permits presenting RTAI's behavior in a control-graph form and presenting statistics regarding the overall system performance (each running real-time tasks can be inspected). System tracing with a very small overhead (claimed to be less than 1 micro-second per event, which seems optimistic considering the expense of time-stamping and the time-stamp imprecision being in the same-range for many if no all systems). Never the less LTT can be applied to production systems as a logging and tracing facility to monitor a real-time system with an acceptable overhead.

Multiprocessor systems and processor features like TSC are supported, cross platform reading is supported. Last RTAI version, found referenced to be supported, 24.1.8, it can be expected that LTT will support more recent versions though (if not already the case - documentation sometimes lags a bit).

http://www.opersys.com/LTT/downloads.html

The RTAI side of the tracer code is in rtai_trace.c which registers and unregisters the trace facility, the trace event macros are from `include/rtai_trace.h` provide a set of entry and exit points that are traced by default: User events are not supported in the rtai_trace facility (though it seems trivial to add them), LTT provides application specific event registration for tracing. (CLEANUP: check in what form LTT provides user events in RTAI).

(TODO: phase 2 tracer overhead)

FaulTolerant Application Monitor, ftappmon The FaultTolerant, FT, application monitor developed in the framework of the OCERA project, named ftappmon, is a higher level analyzes tool that allows runtime monitoring and intervention. The FT application monitor provides a dedicated FT API to

the RTLinux/GPL based rt-application. he FT application monitor is used in conjunction with the FT controller component.

FT-controller, is a low level module capable of intervening on abnormal situations like timing errors, thread abortion. It provides replacement behavior for the faulty thread.

The FT controller for RTLinux/GPL interfaces to the POSIX Trace (see above) by using a kernel stream tracer in order to get system events date which constitutes the basis for process error analysis. The FT controller provides an analyzing and filtering capability (filtering specific event) and on events triggers a decision making instance to decide if normal or abnormal behavior occurred.

http://www.ocera.org/download/components/WP6/ftappmon-0.1-1.html

http://www.ocera.org/download/components/WP6/ftcontroller-0.1-1.html

This monitoring and intervention system is intended not only for analysis purposes during development but conceptually intended for production systems that have excessive monitoring demands.

Visualization

The visualization tools available are used in combination with trace tools (see above). Visualization tools are available for RTAI, ADEOS and for RTLinux/GPL.

crono Now more or less obsolete visualization tool for RTLinux/GPL, use of crono is deprecated as kiwi provides enhanced capabilities.
http://rtportal.upv.es/apps/crono/

kiwi In conjunction with the POSIX tracer, kiwi can visualize scheduling jittery and task switch timing as well as other traced events. Kiwi in principal is not coupled with a specific tracer but implements an event-data format that can be used by any tool including 'home-brew' application traces.

Kiwi is intended for graphical presentation of debugging and run- time monitoring data, so called trace logs, its prime target was to constitute a professional development tool for RTLinux in the frame-work of the OCERA project (where it was successfully utilized to support the porting of GNAT to RTLinux-GPL.

Written in Tcl/TK, it is more or less platform independent, at least with respect to Linux development systems. Its main features are zoom, rich set of graphical elements, output to eps files (directly usable in TeX and other documentation systems), and events driven navigation within the trace-logs. Kiwi is mainly focused on real-time systems, but you can use it for represent any kind of concurrent application or system provided the trace data format is met.

http://rtportal.upv.es/apps/kiwi/

LTT LTT has the visualization tools integrated into the LTT releases (see above).

Chapter 11

Support

11.0.1 Community support

The implementations discussed here are partially under open licenses, for these implementations (RTAI/RTAHL, RTAI/ADEOS, RTLinux/GPL, Mainstream Preemptive Kernel) support is provided mainly by the developer and user-community. The essence of this support is NOT that it is free of charge, the essence is that the feedback provided by these mailing lists is communicating technological know-how and not just 'solutions'. Most commercial support offerings will try to 'black-box' the product, even open-source products can be 'black-boxed' making the know how unavailable, or at least not available with a reasonable effort. The open-source initiatives that lead to realtime extended Linux variants have an open policy with respect to underlaying technologies and encourage the transferee of these technologies.

How do you use open-source support ? The best way to gain know how on an open-source project is to participate in the project, the security of developing the know how for a given technology in-house which has open-sources is considerably higher than can be provided by signing a support contract with a commercial company. One need not be an expert to participate in a project, starting by answering simple questions on the project mailing lists and providing bug-report will give a reasonable good insight into projects for new comers. Note that a substantial part of the problems related to introduction of a new technology are not the scientific questions but the procedural questions, this is one of the strength of open-source projects that these procedural issues are handled in public and not hidden from the end-users.

It should also be noted that there are some tools available provided by the open-source community, like ksymoops for kernel errors, and error reporting mailing lists as well as bug reporting tools (i.e. bugzilla data base interface, sendmails bug-buddy) - when a team starts working with an open-source technology it needs to review available tools as to get the optimal community sup-

port.

11.0.2 Commercial support

For all variants of hard real-time Linux there are commercial support offerings available.

(TODO: contact infos)

Chapter 12

Reference Projects

The number of projects published, utilizing hard real-time enhanced Linux is very long, we will give some pointers to locations for details and then present a few projects that we see as demonstrating the capabilities of these enhancements very well. It should be noted that ADEOS is underrepresented here as it is a fairly new development and not much has yet been published, expect this to change in the near future.

12.1 Information sources

The list below may seem quite beasty as one of the authors was one of the initiators of these events in 1999 and is involved in the preparations of these workshops, but it seems legitimate as this forum constitutes the only dedicated forum at this point to present current developments targeting hard real-time enhanced Linux variants specifically. This should not lead to the impression that there are no other relevant publications, but the intention of this chapter is to provide the reader with an overview of existing efforts and successful projects, for this purpose the presentations at these Workshops can be considered covering the entire spectrum in a reasonably representative manner.

1st. Real Time Linux Workshop, Vienna, Austria, 1999
`http://www.realtimelinuxfoundation.org/events/rtlws-1999/presentations.html`

2nd. Real Time Linux Workshop, Orlando, USA, 2000
`http://www.realtimelinuxfoundation.org/events/rtlws-2000/presentations.html`

3rd. Real Time Linux Workshop, Milan, Italy, 2001
`http://www.realtimelinuxfoundation.org/events/rtlws-2001/papers.html`

4th. Real Time Linux Workshop, Boston, USA, 2002
`http://www.realtimelinuxfoundation.org/events/rtlws-2002/papers.html`

5th. Real Time Linux Workshop, Valencia, Spain, 2003
(to be held November 9-11)
`http://www.realtimelinuxfoundation.org/events/rtlws-2003/papers.html`

12.1.1 Variant specific references

As only RTAI really has a representative selection of projects on its home-page we list it here, the FSMLabs.com web-site for RTLinux/Pro gives little technical project reports (marketing material) and the ADEOS project has not yet published reference projects (this is expected to follow fairly soon though)

- RTAI: `http://www.aero.polimi.it/ rtai/applications/index.html`
- ADEOS: `http://www.adeos.org` (not much there yet)
- RTLinux: `http://www.opentech.at`

(CLEANUP: refs to further project pages)

12.2 Some representative Projects

We here reproduce some selected abstracts of the Real Time Linux Workshops, to present the range of applications that hard real-time enhanced Linux has been utilized for.

12.2.1 RT-Linux for Adaptive Cardiac Arrhythmia Control

Author: *David Christini*

Typical cardiac electro-physiology laboratory stimulators are adequate for periodic pacing protocols, but are ill-suited for complex adaptive pacing. Recently, there has been considerable interest in innovative cardiac arrhythmia control techniques, such as chaos control, that utilize adaptive feedback pacing. Experimental investigation of such techniques requires a system capable of real-time parameter adaptation and modulation. To this end, we have used RT-Linux, the Comedi device interface system, and the Qt C++ graphical user interface toolkit to develop a system capable of real-time complex adaptive pacing. We use this system in clinical cardiac electro-physiology procedures to test novel arrhythmia control therapies.

Comment: This paper was select because it demonstrates the reliability of the hard real-time enhanced variants. The reliability demands for this projects were very hight !

Full Paper: `ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-1999/proc/p07-christini.pdf.zip`

12.2.2 Employing Real-Time Linux in a Test Bench for Rotating Micro Mechanical Devices

Author: *Peter Wurmsdobler*

This paper describes a testing stage based on real time Linux for characterizing rotating micro mechanical devices in terms of their performance, quality and power consumption. In order to accomplish this, a kernel module employs several real time threads. One thread is used to control the speed of a master rotating up to 40000rpm by means of an incremental coder and a PCI counter board with the corresponding interrupt service routine. Another thread controls the slave motor to be tested, synchronized to the coder impulses using voltage functions saved in shared memory. The measurement thread is then responsible to acquire date synchronously to the rotor angle and stuffs data on voltages, currents, torque and speed into different FIFOs. Finally, a watchdog thread supervises timing and wakes up a users space program if data have been put in the FIFO. This GTK+ based graphical users space application prepares control information like voltage functions, processes data picked up from the FIFOs and displays results in figures.

Comment: A good example of utilizing hard-realtime enhanced Linux for equipment testing, especially for small numbers of specialized devices this option is of interest.

Full Paper: `ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-1999/proc/p-a05_peterw.pdf.zip`

12.2.3 Remote Data Acquisition and Control System for Mössbauer Spectroscopy Based on RT-Linux

Author: *Zhou Qing Guo*

In this paper a remote data acquisition system for Mössbauer Spectroscopy based on RT-Linux is presented. More precisely, a kernel module is in charge of collecting the data from the data acquisition card which is self-made based on ISA, sharing the data with the normal Linux process through the module of mbuff, carrying out the remote control and returning the results to the client by building a simple and effectual communication model. It's a good sample to deal with the communication between the real time process and the normal process. This user application can access to this system by the browser, or Java program to implement the real time observation and control.

Comment: even though this paper has some language weaknesses it shows in a very nice way how hard-realtime non-realtime are integrated and other OS-independent technologies (Java,web-interfaces) can be utilized to interface to existing non-UNIX systems.

Full Paper: `ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-1999/proc/p-a09_guo.pdf.zip`

12.2.4 RTLinux in CNC machine control

Author: *Yvonne Wagner*

In this article the project of porting an axis controller for turning and milling machines running under the real time operating system IA-SPOX to RTLinux is described. EMCO implements PC based control systems for training machines with different user interfaces simulating control systems like Siemens or Fanuc. The current real time system IA-SPOX is running under Windows on the same computer as the application, exchanging set and actual values via ISA card with the machine. As the old RTOS is not supported any more under Windows 2000, a new solution had to be found. The reason, why it is now possible to use RTLinux, lies in the flexibility of the whole new control system where the graphical user interfaces and the axis controller are separated. Our GUIs will continue to run under Windows, but communicate with the real time tasks via Ethernet. Thus, the platform for Real Time can be freely chosen without paying attention to the current operating system our clients use. The target will be a motherboard booted from flash disk with RTLinux as its operating system, more precisely miniRTL for our embedded system. The data exchange to the axis will be realized by addressing a PCI card every control cycle.

The points, where porting the axis controller tasks on RTLinux required some redesign, are explained, and also some other problems encountered during this on-going project.

Comment: This paper not only demonstrates the application of real-time Linux for industrial CNC machine tools but also has some valuable inputs on migration from proprietary to open-source systems.

12.2.5 Humanoid Robot H7 for Autonomous & Intelligent Software Research

Author: *Satoshi Kagami*

A humanoid robot "H7" is developed as a platform for the research on perception-action coupling in intelligent behavior of humanoid type robots. The H7 has the features as follows : 1) body which has enough DOFs and each joint has enough torque for full body motion, 2) PC/AT compatible high- performance on-board computer which is controlled by RT-Linux so that from low-level to high-level control is achieved simultaneously, 3) self-contained and connected to a network via radio ethernet, 4) Online walking trajectory generation with collision checking, 5) motion planning by 3D vision functions are available. The H7 is expected to be a common test-bed in experiment and discussion for various aspects of intelligent humanoid robotics.

Comment: Not only a good example of a very complex system utilizing hard real-time enabled Linux in robotics, but also a very amusing thing to watch walking.
Full Paper: `ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2001/proc/k02-kagami.pdf.zip`

12.2.6 Real-time Linux in Chemical Process Control: Some Application Results

Author:*Andrey Romanenko, Lino O. Santos and Paulo A.F.N.A. Alfonso Andrey Romanenko*

Many chemical processes require real-time control and supervision in order to operate them safely and profitably, while satisfying quality and environmental standards. As a means to comply with these requirements, it is common practice to use control software based on a proprietary operating system such as QNX, WxWorks, or MS Windows with real-time extensions. To our knowledge, the idea of using Real-Time Linux has not been embraced widely by research and industrial institutions in the area of Chemical Engineering. Nevertheless, recent application reports from other industrial fields indicate that several variants of the Linux operating system, that enable it to be real-time, are an attractive and inexpensive alternative to the commercial software. In fact, several implementations of open source data acquisition and control software, and real-time simulation environment have been developed recently. Moreover, the apparent trend for the number of such applications is to increase. We describe our experience at the Department of Chemical Engineering of the University of Coimbra, with two pilot plants that are under control of a system based on real-time Linux. One of them is a completed project and the other is under development. The experimental set-ups closely resemble industrial equipment and function in similar operating conditions. We hope that this successful application will encourage further deployment of real-time Linux in the Chemical Engineering research and industry.
Full Paper: `ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2002/proc/a04_romanenko.pdf.zip`

There are a number of other papers that could have fit here ranging from scientific instrument control, flight-simulators to 60MW puls-generators for high-energy physics, as a conclusive summary it can be stated that the hard real-time enhanced Linux variants have been applied to almost any field of industrial and scientific processing.

Part II

Main Stream Linux Preemption

Chapter 13

Introduction

The development relevant for soft-realtime systems in the main-stream kernel was triggered by the move toward SMP support that was really only available beginning with the 2.2.X series of kernels (earlier 2.0.X kernels kind of supported "asymmetric-SMP") which lead to demands for advanced synchronisation mechanisms and kernel threading to improve scalability. With the move beyond dual-CPU SMP systems these demands got an almost dominant position in the kernel development efforts. Changes relevant for soft-realtime have been moving into the kernel slowly - Beginning with the early 2.2.X kernels developments for preemption in kernel context, more specifically in system-calls, began as external patches. Softirq introduction in the 2.3.43 kernel opened the path for efficient DSR implementations,in the 2.5.X series of the main-stream kernel development (unstable development tree) began introducing low latency approaches, order one scheduling (O(1)) and improved timers. These activities and more all improve the soft-realtime capabilities of mainstream Linux, but one should be aware of the fact that the motivation for these introduced concepts is NOT realtime but scalability. The demands for scalability of an OS are:

- fast synchronisation primitives
- fine grain locking
- increased threading
- efficient ISR/DSR coexistence

In the following chapter the changes in the current development kernels and in the early stages of the (to be stable) 2.6.X kernel are described from the perspective of realtime enhanced Linux systems. First we talk about the key issues in the stock kernel, like the flow of time, scheduling algorithms, high resolution timers and last but not least about low-latency and preemption patch. To clarify, if we talk about preemption, than preempting kernel paths are meant, user space processes are fully preemptible and have been in all versions of linux.

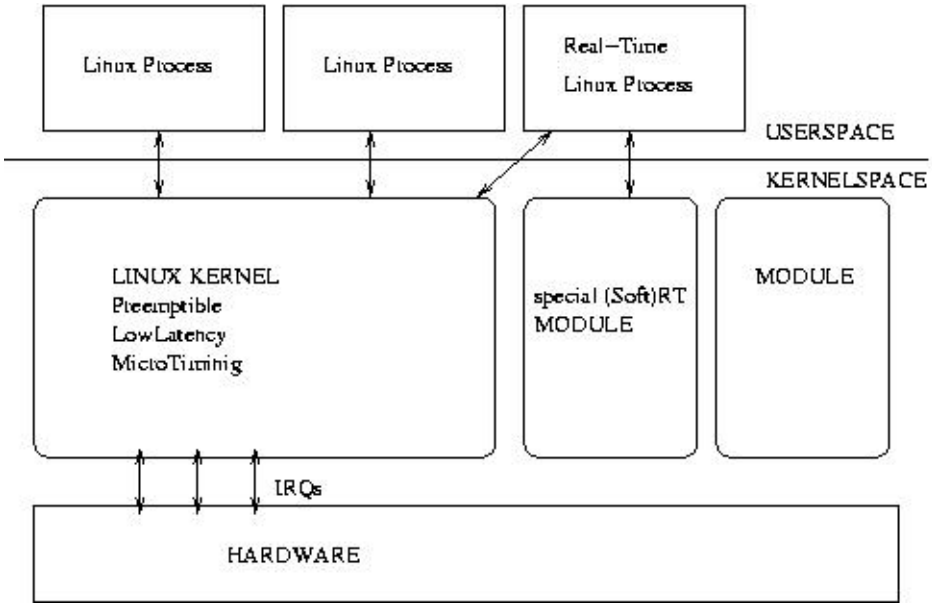

Figure 13.1: Kernel Modification Variants

The figure 13.1 depicts the structure of a modified linux kernel variant. If you compare this picture with the microkernel real-time variants than the real-time processes are still running in the "normal" linux environment in the kernel preemtion approaches. The "special" (Soft) real-time modules are using the new features of the modified kernel and are supporting the soft real-time userspace processes.

Chapter 14

Mainstream Kernel Details

This chapter describes the mainstream kernel in a depper way specifically for real-time demands. First of all we discuss the flow of time in the kernel, that means, timer interrupt, how is time stored, how to handle delays for a specified amount of time and scheduling functions after a specified time lapse. As will see some improvements in time resolution this covers section "High Resolution Timers" are needed. After that come the scheduling algorithms and how they are used in the different (soft)realtime solutions. In the following chapter preemptive and lowlatency patch are explained and how they influence the issues above.

14.1 Time in Mainstream Kernel

As described in [7] two main kinds of timing measurement must performed by the Linux kernel:

- Keeping the current time and date (with `time()`, `ftime()` and `gettimeofday()` system calls thy can be used in userspace applications to read out the values)
- Maintaining timers mechanisms that notifies the user and kernel space programs that a certain interval of tim has elapsed (this is performed with `setittimer()` and `alarm()` System Calls)

Beside these two important timing mechanism we also talk about delaying execution.
The 2.4.x kernel time interval is architecture dependent and is defined by the HZ symbol in `asm/param.h` The symbol HZ specifies the number of clock ticks generated per second. Examples of platform specific values of HZ:

- **i386,arm,ppc,mk68,sh**: 100
- **alpha**: 1024
- **IA64 simulator**: 32

- **IA64**: 1024

As you can see for the most platforms the time interval is set to 1/100 = 10ms, so at every timer interrupt the value of jiffies is incremented. Certainly is it possible to change the value of HZ which changes the interval, that is sometimes done for improving response time. Therefore no driver writer should count on any specific value of HZ, but it is not recommeded as this will break some existing drivers.

14.1.1 Current Time

In kernel code the current time is stored in variable `jiffies`, kernel space can always retrieve the current time by looking at the global variable `jiffies`. The `jiffies` value is incremented every time tick which corresponds to the resolution described above. The interrupte service routine defined in `arch/i386/kernel/time.c` calls, depend on the clock source, the function`do_timer` which is defined in `kernel/timer.c` :

```
void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
```

Be aware that the `jiffies` value represents only the time since the last boot, revered to as epoch in UNIX. This means that you can use `jiffies` directly only to measure time intervals or to calculate the uptime.

14.1.2 Delaying Execution

Long Delays

The best way to do a delay is to let the kernel do it for you, there are two ways of setting up timeouts:

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
```

Short Delays

For short delays then two kernel functions are available:

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

This functions are available for most of the supported architectures and uses software loops for the required number of microseconds. `mdelay` is a loop around `udelay` which is based on the integer value `loops_per_second` as result of the BogoMips caculation performed at boot time. This short time delays are normally used in kernel drivers for various hardware. The udelay macro is used approximately 2500 and the mdelay 500 times in the linux/drivers directory (Linux Kernel 2.4.20). (TODO second phase:
- Are this enormous amount of sleeps(delays), mainly used in kernel drivers, a potential for optimizations ?
- Benchmark jitter amd overhead of mdelay/udelay)

14.1.3 Timers

Interval Timers

The linux kernel allows userspace processes to set special *interval timers* for periodic and not periodic signals. The *itimer* causes Unix signals only once or periodically, depending on the frequency parameter. The characteristics of each interval timer are:

- the frequency (defines the time interval at which the signals must be emitted, if the value is null than just one signal is generated)
- the remaining time (time until the next signal is generated)

The accuracy of these timers is not very high, because it is impossible to predict when the signals will be delivered.
The actually stable kernel does supply BSD-Timers with the following interfaces:

```
long sys_setitimer(int which, struct itimerval *value,
      struct itimerval *ovalue)

long sys_getitimer(int which, struct itimerval *value)
```

The libc in userspace offers following corresponding functions:

```
- int setitimer (int WHICH, struct itimerval *NEW, struct
          itimerval *OLD)

The ‘setitimer’ function sets the timer specified by
WHICH according to NEW.
```

```
The WHICH argument can have a value of 'ITIMER_REAL',
'ITIMER_VIRTUAL', or 'ITIMER_PROF'.

If OLD is not a null pointer, ‘setitimer’ returns information
about any previous unexpired timer of the same kind in the
structure it points to.

The return value is ‘0’ on success and ‘-1’ on failure.  The
following ‘errno’ error conditions are defined for this function:

    ‘EINVAL’
         The timer period is too large.

- int getitimer (int WHICH, struct itimerval *OLD)
     The ‘getitimer’ function stores information about the timer
     specified by WHICH in the structure pointed at by OLD.

     The return value and error conditions are the same as for
     ‘setitimer’.

‘ITIMER_REAL’
     This constant can be used as the WHICH argument to the ‘setitimer’
     and ‘getitimer’ functions to specify the real-time timer.
 The actual elapsed time; the process reveives SIGALRM signals.

‘ITIMER_VIRTUAL’
     This constant can be used as the WHICH argument to the ‘setitimer’
     and ‘getitimer’ functions to specify the virtual timer.
 This is the time spent by the process in User Mode; the process
 receives SIGVTALRM signals.

‘ITIMER_PROF’
     This constant can be used as the WHICH argument to the ‘setitimer’
     and ‘getitimer’ functions to specify the profiling timer.
 Time spent by the process in both; User and Kernel Mode; the process
 decriptor receives SIGPROF signals.
```

As described in [7] the ITIMER_REAL interval timers are using dynamic timers because the kernel has to deliver the signals to the process even when it is not running on the CPU. So each process descriptor includes a dynamic timer object named `real_timer`. The setitimer() system call initializes the `real_timer` fields and after the `add_timer()` function is called ti adds the dynamic timer to the proper list. When a timer expires the SIGALRM signal is send to the process using the `it_real_fn()` timer function (if `it_real_incr`

is not null, it sets the expires field again reactivating the timer).

ITIMER_VIRTUAL and ITIMER_PROF interval timers do not require the dynamic timer method described above. As they are synchronous with respect to task scheduling timers are updated while the process is running and once every tick, and if they expire, the signal is sent to the current process.

(TODO second phase: Benchmark the itimers)

Alarms

The 'alarm' function sets the real-time timer to expire in SECONDS seconds. If you want to cancel any existing alarm, you can do this by calling 'alarm' with a SECONDS argument of zero, obviously the granularity of one second is not very satisfying for many applications. This granularity has hostorical reasons though and is not changed for compatibility reasons. The return value indicates how many seconds remain before the previous alarm would have been sent. If there is no previous alarm, 'alarm' returns zero.

```
unsigned int alarm (unsigned int SECONDS)
```

Example
A demonstrative example is taken out of *info libc*:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* This flag controls termination of the main loop. */
volatile sig_atomic_t keep_going = 1;

/* The signal handler just clears the flag and re-enables itself. */
void catch_alarm (int sig)
{
    keep_going = 0;
    signal (sig, catch_alarm);
}

void do_stuff (void)
{
    puts ("Doing stuff while waiting for alarm....");
}

int main (void)
{
    /* Establish a handler for SIGALRM signals. */
```

```
        signal (SIGALRM, catch_alarm);

        /* Set an alarm to go off in a little while. */
        alarm (2);

        /* Check the flag once in a while to see when to quit. */
        while (keep_going)
             do_stuff ();

        return EXIT_SUCCESS;
    }
```

As shown above the alarm interface is very easy to use, it is important to catch the alarm signal. But if a better resolution than seconds is needed than the alarm function is the wrong solution.

More about timers and its clock sources will be described in the section [?] High Resolution Timers at the end of this chapter.

14.2 Scheduler

The scheduler is responsible for managing the cpu resource allocating it to the different processes. As described in [G.Buttazzo-Hard real-time computing systems]:

> When a single processor has to execute a set of concurrent tasks - that is, tasks that can overlap in time - the CPU has to be assigned to the various tasks according to a predefined criterion, called a *scheduling policy*. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm*. The specific operation of allocations the CPU to a task selected by the scheduling algorithm is referred as dispatching.

The *scheduling policy*, *scheduling algorithm* and the *dispatcher* are important parts of any modern operating system. Linux Systems are designed to reduce the response time for interactive processes, this makes the system subjective faster for the user.

14.2.1 Mainstream Scheduler

Scheduling Classes/Algoritms

Linux supports different POSIX scheduling classes (algorithms), they can be set with the systemcall `sched_setscheduler()`. These three classes are imple-

mented in kernel/sched.c:

- SCHED_OTHER - each POSIX real-time-process has a higher priority than a process is scheduling class SCHED_OTHER
- SCHED_FIFO - a process runs until it gives back CPU or if a process with higher POSIX real-time priority preempts it will run
- SCHED_RR - each process has his timeslice and would be interrupted if the timeslice is consumed or processes with the same priority occur. That means that processes with the same priority are handled in classical round robin order.

For each class a scheduling algorithm is implemented the default algorithm is being SCHED_OTHER.

The SCHED_OTHER algorithm is not specified in the POSIX standard, because it gives freedom to the operating system programmer to implement his preferd algorithm. In the case of Linux it is actually the order one algorithm (or short O(1)), which anticipates to combine two conflicting daemons, maximimum throughput and good response to interactive user.

O(1)-Scheduler

The O(1) scheduler contains two priority-ordered arrays per CPU:

- active array - contains all tasks that have timeslices left
- expired array - holds all tasks that have used up all there timeslices

This arrays are accessed directly over two pointers in the per CPU runqueue structures. If all active tasks are used up then the two arrays are switch, that means, the active array is now the new expired array and the old expired array is now the new active array. So for the active array an arbitrary number of active and expired tasks can be used and easily switch to each other. The ideal solution is to combine this mechanism with roundrobin scheduling and the result is a hybride priority-list and array-switch method of distributing timeslices. The big advantage is to split the complete task list into active and expired list so a portion of task can be processed with a appropriate scheduling mechanism.

from kernel/sched.c

```
*  2002-01-04 New ultra-scalable O(1) scheduler by Ingo Molnar:
 * hybrid priority-list and round-robin design with
 * an array-switch method of distributing timeslices
 * and per-CPU runqueues.  Cleanups and useful suggestions
 * by Davide Libenzi, preemptible kernel bits by Robert Love.
```

TODO - more schedulers?

This is a task for the second phase of the project, because in the opensource world there are more scheduling optimizations around - some specifically targeting real-time. One good description of new or alternative scheduling methods can be found in [15].

14.3 High Resolution Timers

As you have seen the stock (or vanilla) linux kernel from [6] supports only 10ms resolution by default, so the scheduler can only guarantee process switching in 10ms worst case periods.
For many higher technical requirements the standard time resolution is not sufficient and so the high resolution timers project was initiated. With the high resolution timers patch from [8] at least 1 micro second resolutions is achievable.

14.3.1 Overview and History

The project "High Res POSIX timers" at sourceforge.net [8] is to design and code high resolution timers for the linux operating system that conform to the POSIX API. The project aim is to have the resulting code accepted and integrated into the standard linux kernel, but this is actually not happend (actual latest stable kernel is 2.4.22 and latest beta version is 2.6.0-test4).
This project is sponsored by Montavista as a GPL initiative and it adds at least 1 micro second resolution timers. Actually the current POSIX API defines two different timer definitions:

- BSD Timers: setitimer() and getitimer() functions (compare section 14.1.3)
- IEEE 1003.1b REAL-TIME timers: timer_gettimer(), timer_settimer ...

As mentioned above the 2.4.x kernel versions providing BSD Timers and if patched with the high resolution timer patch they support the second POSIC real-time timers to. The kernel 2.6.x will include the IEEE 1003.1b POSIX API to support POSIX REAL-TIME timers by default. The implementation and API can be found in the kernel tree in `include/linux/posix-timers.h` and `kernel/posix-timers.c`.

14.3.2 Design and Implementation

The high resolution timer is not for free, because it adds a small overhead at each time a timer expires. There is no overhead if no high resolution timer is activated. With active hrt option a best case resolution at least one micro second can be provided.

If a linux kernel (2.4.20) is patched with the High Resolution Timer from [8] the following new options are available during kernel for x86 configuration (`make menuconfig`):

```
(3000) System wide maximum number of POSIX timers (NEW)
        [*] Configure High-Resolution-Timers (NEW)
        (Time-stamp-counter/TSC) Clock source?
        (512) Size of timer list?
```

Note: Activating HR-Timers also enables the options POSIX_CLOCKS, CLOCK_REALTIME_HR and CLOCK_MONOTONIC_HR but does not change the resolutions of CLOCK_REALTIME or CLOCK_MONOTONIC they stay at 1/HZ resolution.

As you can see, with the kernel option above, the maximum numbers of POSIX timers and the size of timer list could be modified and should be set to the active application demands. The system wide number of POSIX timers allows you to configure maximum number of POSIX timers. Timers are allocated as needed so the only memory overhead this adds is about 4 bytes for every 50 or so timers to keep track of each block of timers. The system quietly rounds this number up to fill out a timer allocation block. It is possible but not recommended to have several thousand timers as needed by your applications.

From the Kernel description for the timer list size: The list insert time is Order(N/size) where N is the number of active timers. Each list head is 8 bytes, thus a 512 list size requires 4K bytes. Use larger numbers if you will be using a large number of timers and are more concerned about list insertion time than the extra memory usage. (The list size must be a power of 2.)

Clocksources on X86 platforms

Beside the Time-Stamp-Counter(TSC) clock source there are also `ACPI-pm-timer` and `Programable-interrupt-timer/PIT` available as clocksources for high resolution timer project.

As described in the kernel configuraten help:

TSC
The TSC runs at the cpu clock rate (i.e. its resolution is 1/CPU clock) and it has a very low access time on most systems. However, it is subject, in some processors, to throttling to cool the cpu, and to other slow downs during power management. If your cpu is not a mobile version and does not change the TSC frequency for throttling or power management this is the best clock timer.

This small userspace example shows the how the `rdtscl()` macros work and how the TSC value can be read out:

```
unsigned long start,end;

printf("test timers1: with rdtscl makro\n");
rdtscl(start);rdtscl(end);
printf("time lapsed: endtime: %u starttime: %u \
              end-start: %li\n", end, start, end-start);
printf("test timers1: with get_cycles\n");
```

output if executed on a P4/1.8Ghz machine:

```
test timers1: with rdtscl makro
time lapsed: endtime: 4173334012 starttime: 4173333932 end-start: 80
test timers1: with get_cycles
```

ACPI-pm-timer
The ACPI pm timer is available on systems with Advanced Configuration and Power Interface support. The pm timer is available on these systems even if you don't use or enable ACPI in the software or the BIOS (but see Default ACPI pm timer address). The timer has a resolution of about 280 nanoseconds, however, the access time is a bit higher that that of the TSC. Since it is part of ACPI it is intended to keep track of time while the system is under power management, it is not subject to the frequency problems of the TSC.

PIT
The PIT is used to generate interrupts at a preset time or frequency and at any given time will be programmed to interrupt when the next timer is to expire, or latest on the next 1/HZ tick. For this reason it is best to not use this timer as the wall clock timer. This timer has a resolution of 838 nano seconds due to its legacy requency of 1.19MHz. This option should only be used if both ACPI and TSC are not available.

As also described in the configuration help the TSC clocksource is the preferred best way to utilize high resolution timers, because it runs on cpu clock speed. Due to the 64 Bit size of the Time-Stamp-Counter Register the acces to this value is slower and is accessed the following mechanism and can be found in `kernelsource of tsc access` on x86 architectures. The amount of time access for a ppc architecture depends on the access time which is ... than the TSC-Register access.

Kernel API

The HRT patch also adds a POSIX timer API to the kernel independent if the high resolution timers are active or not. Following system call functions are added:

```
int sys_timer_gettime(timer_t *timer, struct itimerspec *cur_setting);

int sys_timer_settime(timer_t *timer, int flags,struct itimerspec
*new_setting, struct itimerspec *old_setting);

int sys_timer_create(clockid_t which_clock,sigevent_t *timer_event_spec,
timer_t *created_timer_id);

int sys_clock_settime(clockid_t which_clock,const struct timespec
*tp);

int sys_clock_gettime(clockid_t which_clock, struct timespec
*tp);
```

14.3.3 Summary

The High Resolution Timers provide micro second resolution and add the POSIX timer API. The POSIX API is an advantage for 2.4.x series, because the mainstream kernel "only" supports the BSD-Timer and the fact that in the 2.6.x kernel series the POSIX timers are included by default, the reuse of development for both kernel series is possible (also for backports).

Todo

This are some todo points for second phase of this study:

- highres allows time resolution down to 1ns, but is the OS able to handle this. Tests are needed.
- compare clocksource access speeds for different architectures, e.g. x86 with ppc access, as well as different clcok sources
- benchmark timers especially with many active timers

Chapter 15

Kernel Preemption in Mainstream Linux

This chapter covers the preemptive and the low latency patches developed an maintained by the open source community. As often seen in the open source linux community there are more than one solution or project targeting the same technical problem or it seems to be handling the same problem. It seems that the Preemption and the Low-Latency-Projects are solutions to minimize the Linux scheduling latency problem. But they come from different areas: the preemption patch was initiated to increase scaleability and the low latency patch comes from the audio community. But as you can see in [16], which describes a unified patch, if you mix the two projects together you probably get the best solution for the problem.
To identify response times and latencies in the Linux Kernel is the main issue for optimizing the kernel responsivness. The kernel response time is the time between the application request and the response from the kernel.
There are four main response time components (in order of time delays, starting with the longest):

- Scheduling Latency (hundreds milliseconds)
- IRQ Handling duration (low hundreds microseconds, depending on implemention)
- Scheduling duration (microseconds)
- IRQ Latency (microseconds, e.g. x86 10us)

As Ingo Molnar mentioned, the maximum scheduling latency is about tens to hundreds of milliseconds, where interrupt latency for x86 hardware is normaly 10us and interrupt duration time is tens to hundreds of microseconds, and last but not least the scheduling duration needs some microseconds. Basically the

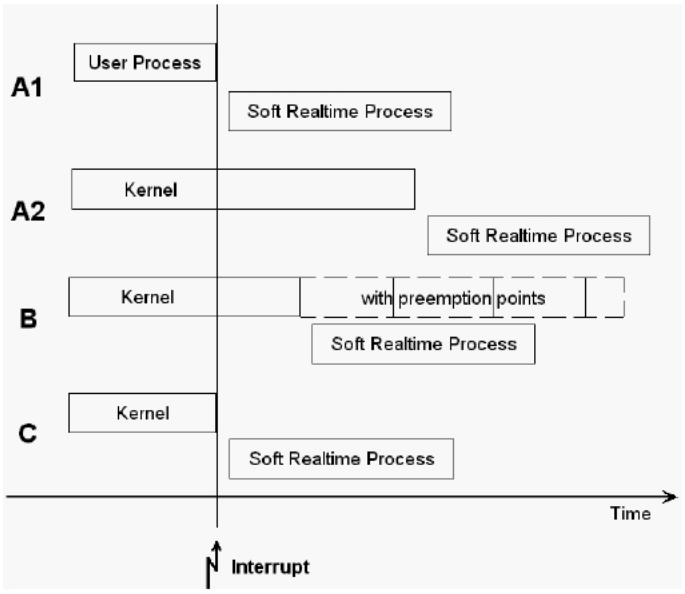

Figure 15.1: Softrealtime Concepts

system worst case response time correlates with the longest kernel path executed automatically.

Figure 15.1 shows the different concepts to make the kernel more responsive. A1 shows the situation if only one userspace process is running, than the lower priority user process will preempted within only some microseconds. But this behaviour is completely different e.g. if a system call which executes non-preemptible kernel code so the process with higher priority has to wait until the systemcall has completed (compare line A2 in figure 15.1. A1 and A2 shows the situation in standard linux kernels at the moment. On the contrary, mentioned above there are two different concepts that try to lower the latencies:

- insert of preemption points into the kernel, system calls are interrupted at special points (Low-Latency Patch, see also B in figure 15.1)
- trying to make the kernel code preemptible in general (Preemption Patch, compare C in figure 15.1)

Both patches have gotten wide testing by now, and can be considered as stable.

15.1 Preemptible Kernel

15.1.1 Overview and History

To improve the responsivness of the standard Linux kernel MontaVista initiated two opensource projects: the Kernel Preemption and a real-time scheduler. In

September 2000 MontaVista unveils there fully preemptible kernel prototype to the opensource community
(article at `http://www.linuxdevices.com/news/NS7572420206.html`) after that Robert M. Love maintains the preemption patch on
`http://www.tech9.net/rml/linux/`. Since kernel version 2.5.4-pre6 the Preemption Patch is an official part of the linux kernel and it is included in the actual 2.6.0-test4 version of the mainstream kernel (and in the stable tree, once the test extention is dropped).

15.1.2 Design and Implementation(Modification) Details

Montavista's concept uses the kernel lock concept which already has been developed for computer systems with SMP (Symmetric Multiprocerssing Systems) on computer systems with only one CPU. The Preemptive Kernel allows a scheduler call after each interrupt. This method does not satisfy hardreal-time demands but it reduces delays for softrealtime processes and increases the responsivness. Also, due to the fair scheduling policy of linux even with preemption linux is inherently non-real-time by design. The "Preemption Patch" for 2.4.x or "Preemptable Kernel Option" for 2.5.x (as of version 2.5.4-pre6) and 2.6.x makes the Linux Kernel interruptable for processes except when the kernel is executing the following

- handling an interrupt
- while SW-interrupt and Buttom Half/Tasklets
- executing the scheduler himself
- while intializing new processes with the *fork()* system call
- during spinlock, writelock or readlock holding[1](This methodes are used in the kernel to protect the kernel in case of Symmetric Multiprocessing, and makes the kernel not preemptible and reentrancy too)

The preemption method is critical in SMP machines and the following issues must be taken care of:

- per CPU data structure need explicit protection
- CPU state must be protected
- Lock acquire and release must be performed by the same task
- Lock hold times must be short

This and some more minor reasons makes it necessary to modify the kernel source with the two functions *preempt_disable* and *preempt_enable* at proper points. In any other times the patch/option allows preemption.

[1]Especially this locks are critical to new "in-house" developments

The kernel patch/option (2.4.x/2.5.x and 2.6.0) Details

This short text describes the kernel option PREEMPT (2.5.65 Kernel) which can be considered valid at least for the current 2.6.0-testX series of kernels:

```
Preemptable Kernel (PREEMPT)

This option reduces the latency of the kernel when reacting to
real-time or interactive events by allowing a low priority process to
be preempted even if it is in kernel mode executing a system call.
This allows applications to run more reliably even when the system is
under load.
```

If this option is activated than the kernel would be build as preemptible version. This option modifies or activates code described in the following parts.

The preemtible kernel patch/option has a directly relation to SMP spinlocks, which are fundamental to Linux for symetric multiprocessing systems.

For the preemtible kernel these four parts are needed changes(as detailed in [14]):

- definition and implementation of a spinlock
- interrupt handling sw to allow rescheduling on return from interrupt if a high prior process becomes executable
- spinlock unlocks, to return into a preemptible system
- kernel build definition for uniprocessor machines must adopted to include preemption spinlocks

The modified `spin_lock()` macro calls the `preempt_disable()` first and changes the spinlock variable. `spin_unlock` A variable `preempt_count` is added to the task structure and the macros `preempt_disable()`, `preempt_enable()` and `preempt_enable_no_resched()` modifies the `preempt_count` field. And helps to prevent preemptions when the system enters one of the exeptions described above. That means a Preemption-Lock-Counter is incremented and indicates if it is forbidden to preempt the kernel code, or not. The lock is released if the system is leaving the exeptions[2] and the Preemption-Lock-Counter is decremented. Another test is done after leaving this regions, namely a check if there was a meanwhile preemption. So a Preemption Point is also included there.

[2] IRQ handling, Sw IRQ, executing scheduler, init of fork and during spinlock, writelock, readlock holding

Script	without patch	with patch
Find script	78.51ms	0.48ms
Launch script	0.61ms	0.41ms
File move script	0.61ms	0.31ms

15.1.3 Some Test Results

Following are some, not verified, test result from different resources found on the web, these results are present without any validation - as they are to be seen as preliminary. They should only give a feeling for the increase in responsivness no more.

Realfree Test

The realfree test program is described later in the appendix, here are just the results. The realfeel program measures the interrupts response times from interrupt to interrupt. Table XYX shows the test results which were published in [19] a general report on real-time enhancements in linux focussed on interrupt-latency. The find script searches a file on hard drive, the launch script runs countinually launches of a trivial program and the move scipt moves continually copies of two large files over each other[3].
Testbed:

- PowerPC
- Linux Kernel 2.4.19
- Preemption Patch from XYX (Robert Love Page)
- 312.5MHz

The patch reduces the interrupt latency for the find script more the 100 times and for launch and move script about 50% (compare **??**). The test takes over 3.5 millons samples and the results are shown in figure 15.2.

RhealStone Test

This test was published at the Real-Time and Embedded Computer Conferenc in Milan. For a complete intro to RhealStone Benchmarks see **??**. Here is a short short description of the benchmark:

- ...
- ...

[3] no disc was connected, the test used network filesystem! Todo for second phase: validate this results with harddisk

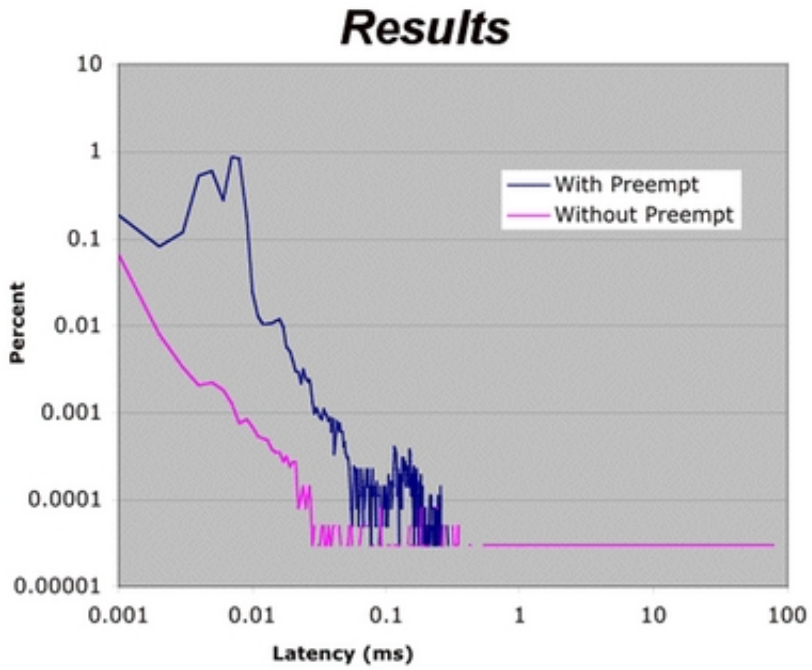

Figure 15.2: Histogram of Latencies [?]

15.1.4 Summary

The preemption patch/option is easier to maintain than the low latency patch, because it is bound to SMP facilities and there requirements - that means that it is integrated into a stable smp kernel mechanism offering additional benefit of making applications well scaleable. Instead the lowlat patch guys must always identifying the sources of delays in a new kernel version, or have to check if there is a patched part that has changed.
The preembtible kernel patch reduces the interrupt latency time dramatically and moves a standard linux system towards a real-time operating system.
But there are also some disadvantages because some tests [?] have shown that the preemption patch/option introduces a relevant performance penalty. To verify this, tests, which are sometimes not so objective and descriptions of the testbeds are missing, some indepentantly tests are needed. Clearly a big advantage of the preemption methode is that it is integrated into the new kernel version 2.6 and thus not only gets much testing but is expected to be well maintained.

15.1.5 Notes

critical thougths

There are still some problems during device initialization code that assumes non-preemption but this could be fixed with disabling the preemption during this time with the *preempt_disable* and *preempt_enable*[4].
Some are oppose a preemptible kernel because of code complexity and throughput interests. The argument of code complexity false, because the preemption patch uses the advantage of already required and in place SMP locking, so no additional complexity is created and linux kernel engineering must already keep in mind the SMP requirements[5].

Which is it: -ible, or -able?

As Rick Lehrbaum wrote at wwww.linuxdevices.com there it is not clear which word should used:

- Preemptable
- Preemptible

They decided to use preemptible, but in the open source community preembtable is more familiar. But as Rick mentioned: "But is popularity really the best measure of what's right?" We prefer preemptible, because www.linuxdevices.com decided to use ible.

15.2 Low Latency Option/Patch

15.2.1 Overview and History

The low latency patch is written to reduce latency for audio applications (streaming and multimedia) so the thresholds are given from this class of applications. The measureable latency becomes perceptible typical at 7msec latency time and should be acceptable for normal audio desktop applications (see []). Up to 5msec should be considered as ideal platforms. The low latency patches enables platforms to be under 4ms, so a low latency patch linux system can be used for professional midi syntisizers with a range between 2-5ms.
Ingo Molnar started in 1999 with identifying and patching the kernels 2.2.10 after 2.4.2 the low-latency project was taken over by Andrew Morton and is still maintained by him (latest patch version 2.4.21).

[4]Note: Premptive Kernels requires that the drivers are premmption aware
[5]it does require SMP core code is not influenced by the patches

15.2.2 Design and Modification

In case of the kernel is not designed for preemption points it is a high critical and enormously sensible to insert preemption points. The Low Latency patch does this and sets explicit preemption points e.g. in places which are iterate over large data structures and consume a lot of time in handling this stuctures. As you can imagine it takes a lot of maintenance because the dynamics nature of the linux kernel development project makes it hard to follow. There are some support [?] tools to identify or find potential places for preemption points but it is just one thing to find them than you have to examine the logic behind the code block and the preemption must set carefully.
As Ingo Molnar started the lowlatency patch project he identified six low latency sources:

- Calls to the disk buffer cache
- Memory page management
- Calls to the /proc file system
- VGA and console management
- The forking and exits of large processes
- The keyboard driver

The lowlatency patch adds the include file `low-latency.h` in include/linux kernel directory, in this file some macros and function prototyps defined extern like `ll_copy_to_user()`, but most changes are done in `kernel/sched.c`.

Preemption Points

To reduce the long latency times above it is necessary to insert preemption points. For preempting a task the functions `conditional_schedule_needed()`, `unconditional_schedule` and `conditional_schedule()` are used. A minimal Preemption Point looks like:

```
if (current->need_resched)
{
    current->state = TASK_RUNNING;
    schedule();
}
```

More complicated examples are the best way to show how a preemption points work and how to insert them. The file `fs/inode.c` is modified by the low latency patch in the function `invalidate_list()`:

```
/*
 * Invalidate all inodes for a device.
 */
static int invalidate_list(struct list_head *head, struct super_block * sb, struct list_
{
    struct list_head *next;
    int busy = 0, count = 0;

    next = head->next;
    for (;;) {
        struct list_head * tmp = next;
        struct inode * inode;

        next = next->next;
        if (tmp == head)
             break;
        inode = list_entry(tmp, struct inode, i_list);
>
>       if (conditional_schedule_needed()) {
>           atomic_inc(&inode->i_count);
>           spin_unlock(&inode_lock);
>           unconditional_schedule();
>           spin_lock(&inode_lock);
>           atomic_dec(&inode->i_count);
>       }
>
        if (inode->i_sb != sb)
             continue;
>       atomic_inc(&inode->i_count);
>       spin_unlock(&inode_lock);
        invalidate_inode_buffers(inode);
>       spin_lock(&inode_lock);
>       atomic_dec(&inode->i_count);
        if (!atomic_read(&inode->i_count)) {
            list_del_init(&inode->i_hash);
            list_del(&inode->i_list);
            list_add(&inode->i_list, dispose);
            inode->i_state |= I_FREEING;
            count++;
            continue;
        }
        busy = 1;
    }
    /* only unused inodes may be cached with i_count zero */
```

```
    inodes_stat.nr_unused -= count;
    return busy;
}
```

The lines quated with $>$ are added by the low-lateny patch (2.4.20-low-latency.patch.gz from [18] against kernel 2.4.20 from [6]). As you can see the function iterates over the complete inode list, this is for real-time applications a to long delay, so a preemption point is inserted.

Another example is take out of `fs/ext2/inode.c` to visualize the use of `TEST_RECHED_COUNT(n)` macro. This macro increments the variable `resched_count`, which is defined with the macro `DEFINE_RECHED_COUNT`, and if it greater then `n` the macro returns true and the `if` statement body is executed and a `conditional_schedule()` preempts the task.

```
static inline void ext2_free_data(struct inode *inode, u32 *p, u32 *q)
{
    unsigned long block_to_free = 0, count = 0;
    unsigned long nr;
>   DEFINE_RESCHED_COUNT;

>   for ( ; p < q ; p++) {
>       if (TEST_RESCHED_COUNT(32)) {
>           RESET_RESCHED_COUNT();
>           conditional_schedule();
        }
        nr = le32_to_cpu(*p);
        if (nr) {
            *p = 0;
            /* accumulate blocks to free if they're contiguous */
            if (count == 0)
                goto free_this;
            else if (block_to_free == nr - count)
                count++;
            else {
                mark_inode_dirty(inode);
                ext2_free_blocks (inode, block_to_free, count);
            free_this:
                block_to_free = nr;
                count = 1;
            }
        }
    }
    if (count > 0) {
        mark_inode_dirty(inode);
```

```
        ext2_free_blocks (inode, block_to_free, count);
    }
}
```

TODO: explain the difference between conditional and unconditional schedule

15.2.3 Summary

Ingo Molnars (linux kernel version 2.2) and Andrew Mortons (linux kernel version 2.4) patches have shown that the changes in the kernel can lower the several long latencies down to the order of 5 to 10 milliseconds [?]. But from our point of view there are problems with the concept of lowlatency preemption points. Although it seems that the lowlatency patch actually not affect the stability of the linux kernel, it is nearly impossible to guarantee the total correctness, because no-one can test all execution paths of the kernel. As mentioned above, badly positioned Preemption Points could cause system crashes of the kernel in case of data inconsitencies. To take in considerations this objections it takes time and might be difficult to apply changes in all different kernel regions. And if you look the dynamic kernel development than it makes not easier to follow the mainstraem kernel if it is not in the official kernel tree.

15.2.4 Guidelines

- **Webresource** - http://www.zip.com.au/ akpm/linux/schedlat.html
- **Licensing** - GPL
- **Availability** - Source code available
- **Activity** -
- **Development Status** -
- **Supported OS** - GNU/Linux
- **Kernel version** - Latest Kernel Version 2.4.21
- **Latest Version** - 2.4.21
- **Supported HW-Platforms** - i386, ?
- **Support** - Opensource Community
- **Dates** -
- **Number of active Maintainer** -
- **Performance** -

- **Applications** -
- **Documentation Quality** - good

15.3 TODO

- insert Test Results here
- summary
- features
- guidelines

Chapter 16

Preemptive Linux (Soft)Real-Time Variants

16.1 KURT

16.1.1 Overview and History

The Kansas University Real-time Linux extention (KURT) is a kernelpatch for standard linux kernel (actually available for 2.4.18 kernel). The originator and project leader is Dr. Douglas Niehaus who is head of a group students working at ITTC (The Information and Telecommunication Technology Center, University of Kansas). The mailing list of KURT started in January 1998 and is low actually low active (approximatly 5-20 mails per month). The project was started 1997 with the first patch for kernel version 2.0.34.
KURT supports microsecond resolution an soft realtime scheduling capabilities (compare [?]). KURT includes the UTIME-Patch which patches the kernel time resolution, the original UTIME patch can be found at [10].

16.1.2 Design and technical Details

Timebase - UTIME

As we have seen in 14.1 section the standard linux kernel timers offers 10ms resolution. The utime, or 'micro-time', adds to the kernel microsecond timers. This is done by reprogramming the timer chip to generate interrupts. For this new resolution two fields (*usec* and *flags*) are added into *timer_list* data structure of the kernel (compare in kernelcode `include/linux/timer.h`).

Scheduler

The KURT Kernel Patch adds following scheduling (real-time) policies for Kernel-Mode:

- **focussed**
- **preferred**
- **mixed**

A Process could assigned to one of this three toplevel scheduling modes:

- explicit
- anytime
- periodic

if a process is assigned to explicit mode than one of this submodes must be selected too:

- EPISODIC
- CONTINUOUS

The Normal mode could also selected, that is the default mode from the standard Linux system with the addon of microsecond resolution from the UTIME-Patch. In the **focussed** real-time mode only KURT processes may run and prevent all non-real-time tasks from running. The **prefferd** mode prefers real-time tasks, if there is no real-time task the regular linux kernel scheduler is called, which selects one or switches to the idle-task. The third mode is the **mixed** scheduling which is a mix of two former modes, the different between preferred and mixed is, that mixed gives the processes which are assigned to the anytime class no precedence over non-real-time task. So they are effectivley considered as non-real-time under mixed mode.

Configuration and API

To configure and control kurt a pseudo device is used, which provides three operations: *open*,*close* and *ioctl*. With these methodes and an Application Programming Interface for these three categories:

- general and utility operations
- process initialization, registration and control
- control of the scheduler

16.1.3 Summary

Conclusion

KURT is soft-realtime. it is usable if you have softrealtime processe with microsecond resolution. But if deterministic interrupt response time is needed or for not so hard real-time demands it could be useful. The development of kurt is low active.

Features

- Soft, or Firm, real-time system (Firm is the nomenclature that the KURT team prefer)
- dedicated Kernel Modes firm real-time, which can be switched with well documented API over KURT Pseudo Device
- increasing Time Resolution with `UTIME`
- Task are dynamically loadable modules, so they have direct access to kernel services
- Firm real-time tasks can use standard Linux features and services

16.2 Montavista Linux

16.2.1 Overview and History

Since 1999 Montavista Linux developed a complete Linux-based embedded deployment platform which is optimized to target modern embedded applications. Montavista Linux Professional Edition supports 7 microprocessor architectures with 24 CPU core variants and tool chains and up to 70 board support packages and system reference platforms. In Sep, 2000 MontaVista Software announced at www.linuxdevices.com [12] that they support "hard real-time fully preemptable Linux kernel prototype," based on Linux kernel 2.4.

MontaVista offers three industry/application targeted editions of MontaVista Linux:

- MontaVista Linux Professional Edition
- MontaVista Linux Carrier Grade Edition
- MontaVista Linux Consumer Electronics Edition

MontaVista Linux Professional Edition:
This Edition from MontaVistas embedded operating system and cross development environment is the main product. It provides a common source and binary platform across a broad range of processor architectures. The Professional Edition is the base product for the other two Editions and could be downloaded from [13].

MontaVista Linux Carrier Grade Edition:
This product is the industry standard COTS (Commercial-Off-The-Shelf) Carrier Grade Linux platform providing functionality specifically for Telecom and Datacom with high availability, hardening and real-time performance.

MontaVista Linux Consumer Electronics Edition:
The latest addition to MontaVista Software's product line is the worlds first embedded Linux product targeted at advanced consumer electronics devices. It combines new functionality and tools with rich support of reference platforms to enable the rapid development of a wide range of consumer electronics products.

Montavista sponsors the Preemptible Kernel Project which is maintained by Robert Love who is also working at Montavista. Another project sponsored by MontaVista is the High-Resolution Timer project at sourceforge.net [8]. MontaVista has reconiced that only with the support from the opensource community a maintainable Embedded Linux System could up to date.

16.2.2 Design and Technical Details

Due to the sponsoring of the opensource community with montavistas key technologie issues the preemption patch, the real-time scheduler and the high resolution timer project are under GPL and the main development is done by the opensource community. The Advantage of an Embedded Linux Distribution is the package of development tools, prebuild filesystems and various configuration tools. Following the Montavista Linux Professional Edition (former name was HardHat Linux) is described.

Timebase

As descibed above Montavista supports the High Resolution Timer opensource project. So all of this key technology can be read in chapter **??**.

The Preemptible Linux Kernel

This main technology is since September 2000 also an opensource project, so montavista get help from the community and is so able to include an up to date preemtible kernel into there distribution. More can be found in section 15

Scheduler

Montavista Linux includes a real-time-Scheduler which replaces the standard Linux Scheduler. This Montavista add on increases also the maximum number of available prorities to 2047 while standard real-time priorites range from 1-99.

Development Environment

The Montavista Development Environment includes GPL based projects like: compilers, linkers, make, and other language utilities, specially configured for cross compilation and building of embedded Linux applications. For debugging

the distribution includes the Data Display Debugger (DDD) front-end in combination with gdb, which are executed from the host system. The GDB and DDD support:

- Setting breakpoints and single-stepping
- C/C++ source and assembly views
- Expression evaluation and data structure browsing
- Call stack chain browsing
- Network and Serial debug interfaces
- Shared library debugging
- Debugging device drivers

KDevelop

For the coding process the Montavista uses the kdevelop Integrated Development Einvironment (compare www.kdevelop.org) and with MontaVista's existing gcc and gdb-based cross-compilation environment. Kdevelop also includes Source Code Management with intgrated CVS client.

The Linux Trace Toolkit

The Linux Trace Toolkit Version from Montavista builds on Karim Yaghmour's opensource Linuc Trace Toolkit.
It is a graphical display programs to extracts, and interprets execution details and also enables users to log and analyze processor utilization and allocation information over specified periods of execution, including comprehensive listings of probed events. The Toolkit offers the cross development kernel tracing tool for IA-32/x86 and PowerPC processors. The LTT can be downloaded from `http://www.opersys.com/LTT/`.
More about Linux Trace Toolkit can be found in section 17.2.1.

Target Configuration Tool

To get a right-size of Linux kernel and populate embedded Linux deployment images with an optimal file system, MontaVista introduces the Target Configuration Tool (TCT). This GUI-based utility enables developers to select only needed modules and drivers for inclusion in kernel builds, allowing bootable foot-prints scaled below 500 Kbytes. Using TCT avoids the drudgery of hand editing configuration and make files while managing dependencies among components. The TCT lets developers choose pre-built packages for inclusion in an embedded file system, including system binaries and data as needed.

Library Optimizer Tool

After a correctly build of kernel and filesystem the Library Optimizer Tool can be used to analyze and minimize the size of the shared libraries.

Filesystem

The following list lists the included main software tools in the Montavista Distrubution.
Target Filesystem:

- linux kernel 2.4.18
- glibc-2.2.5
- busybox-0.60.2
- syslinux-1.62
- tinylogin-0.80
- thttpd-2.21
- netkit-base-0.17
- netkit-telnet-0.17
- gdb-5.2.1 (gdbserver)

Development Host:

- gcc-3.2
- binutils-2.12.1
- gdb-5.2.1

Guidlines

The Guidlines moved to chapter **??**.

16.2.3 Notes

Downloaded Preview Kit for IBM405GP only supports: Red Hat 7.2, Mandrake 8.1, SuSE 7.3 and it was hard to find a distribution with such outdated version. It was not possible to install it on a RH9.0 system. So this is always a problem with prebuild embedded development systems, if there is no build script than you also have to store a distribution cd set with the distribution, because years later it could be a problem to find outdated versions of standard distributions.

16.3 TimeSys RTOS

After more than one contact with timesys it was not able to get any useful information for this study, so be careful with the following data. Due to getting no data and technical description from timesys here is just the technical bulletin from the webpage (for TimeSys Linux RTOS Professional Edition):

- Royalty-free real-time capabilities that transform Linux into a real-time operating system (RTOS)
- TimeSys Linux Board Support Package (BSP)
  - TimeSys Linux GPL kernel, based on the 2.4.18 Linux kernel, that delivers:
    - Full kernel preemption
    - Unlimited process priorities
    - Enhanced schedulers
    - Priority schedulable interrupt handlers and Soft IRQs
    - High Availability/Carrier Grade features
    - POSIX Message Queues
  - Lowest-latency Linux kernel on the market
    - Over 100 packages for installing thousands of root filesystem development, debugging, monitoring and management applications and libraries.
    - Complete driver support for the target board
  - Unique priority inversion avoidance mechanisms (Priority Inheritance, Priority Ceiling Emulation Protocol)
  - High resolution timers
  - True soft to hard real-time predictability and performance
  - Available ready-to-run on more than 65 specific embedded development boards spanning 8 processor architectures and 35 unique processors
  - Optional TimeSys Reservations, which guarantee fine-grained performance control of your CPU and network interface, regardless of system overload
  - Certified GNU toolchains for Windows and Linux development hosts
  - Powerful, multi-threaded local and remote debugging with gdb
  - Detailed user and API documentation
  - Superior TimeSys customer support
- TimeStorm, a graphical Integrated Development Environment (IDE):

  - Integrated remote multi-threaded debugging
  - Extensive target interactive support
  - Broad Makefile management
  - Support of multiple cross-platform plug-in compilers
  - Comprehensive source code editor
  - Integration with popular source code control systems
  - Project creation wizards for multiple project types
  - Low cost, easy-to-use package

- TimeTrace, a graphical analysis and visualization package:

  - Detailed target profiling
  - Detailed thread level, process level, and context switch information
  - Enable and disable OS and user event labels
  - Viewable interrupt and task switch statistics
  - Integrated distributed monitoring
  - View the status on all target hardware simultaneously with a single monitoring station
  - Connect and enable targets dynamically
  - Low-cost, easy-to-use package

- Toolchain:

  - gcc/g+ 3.2
  - gdb 5.2.1
  - glibc 2.2.5
  - binutils 2.13
  - KGDB 5.2.1

16.4 Others

Here we list the not so important respectivly not so wide known softrealtime patches for the linux kernel.
TODO maybe we have forgotten some one ;)

Chapter 17

Appendix

17.1 Benchmarks

This section descibes some benchmark tools and results found during scanning throu the web. Be careful, none of tests was verified and they should only show a feeling about the improvements of the patches described above.

17.1.1 Latencies of Linux Scheduler

The scheduler is an important part of each operating system, because it is invoked very often. To get a better feeling which latency times the scheduler produces the benchmarks from [?] are printed here. The test is a little bit outdated because the test was with kernel 2.4.0-test6, but as already mentioned, it is here for giving us some feeling.

The testbed:

- Processor AMD K6 400MHz
- One soft-realtime process running in SCHED_FIFO
- many load processes in SCHED_OTHER policy
- Kernel 2.4.0-test6

The source code of SCHED_FIFO process:

```
int main(int argc, char ** argv)
{
int i,j,ret;
double k=0.0;
struct sched_param parameter;

parameter.sched_priority = 53;
```

```
ret = sched_setscheduler(0, SCHED_FIFO, &parameter);

if (ret == -1){
perror("sched_setscheduler");
exit(1);
}

for(i=0;i<200;i++)
{
//just to run a while
for(j=0;j<50;j++) { k = sqrt(2.0);}
sleep(1); // sleep one second and invoke scheduler
}
return 0;
}
```

It is only one SCHED_FIFO soft real-time process in the system and all other processes are using the standard Linux scheduler policy SCHED_OTHER. The source code of SCHED_OTHER processes looks like the following:

```
int main(int argc, char ** argv)
{
int result;
while(1) { result = sqrt(400); }
exit(0);
}
```

17.1.2 Rhealstone

The Rhealstone metric consits of quantitative measurements of six components that affects the real-time performance of a computer system. (see also [4])

- Task Switching time (t_{TS}): The average time the system takes to switch between independent and active tasks of the same priority
- Preemption time (t_P): Average time it takes to transfer the control from a lower priority to a higher priority task
- Interrupt latency time (t_{IL}): The time from CPU recieves an interrupt to execution of the first instruction in the interrupt service routine.
- Semaphore shuffling time (t_{SS}): The delay in the OS before the task aquires a semaphore that is in the possesion of another task.
- Deadlock breaking time (t_D): This is the average time to break a deadlock caused when a high priority task preempts a low-priority task that is holding a resource the high-priority task needs.

- Datagram throughput time (t_T): Bytes/sec sent from one task to an other task using the system communication primitives.

The Rhealstone metric is calculated as follows: $Rhealstonenumber$

17.1.3 realfeel

TODO: cleanup here

Realfeel is written by Mark Hahn and can be downloaded from For this work, interrupt latency is measured with an open benchmark called Realfeel, written by Mark Hahn. Realfeel issues periodic interrupts and measures the time needed for the computer to respond to these interrupts. Response times vary from interrupt to interrupt. Realfeel measures these interrupts and produces a histogram by putting the measurements into bins. The measurement focused on is not the histogram itself but the largest interrupt latency measurement. The performance of an operating system may depend on the average interrupt latency for some applications, but real-time applications are more dependent on the largest interrupt latency. The largest interrupt latency is a prediction of the worst case scenario. For many of these applications, if the latency were over the limit one time, it would result in a complete failure of the system. So the purpose of the benchmark is to find the latency that would never be exceeded.

17.1.4 TimePegs

17.2 Trace and Debugging Tools

17.2.1 Linux Trace Toolkit

With the LTT it is possible for the kernel to log important events to a tracing driver. For this is a kernel patch is needed and if enabled the use of the generated traces in order to reconstruct the dynamic behavior of the kernel, and hence the whole system is possible.
The tracing process contains 4 parts :

- The logging of events by key parts of the kernel.

- The trace driver that keeps the events in a data buffer.

- A trace daemon that opens the trace driver and is notified every time there is a certain quantity of data to read from the trace driver (using SIG_IO).

- A trace event data decoder that reads the accumulated data and formats it in a human-readable format.

If the kernel patch is enabled the first part of the tracing process will always take place. That is, critical parts of the kernel will call upon the kernel tracing function. The data generated doesn't go any further until a trace driver registers himself as such with the kernel. The tracekit driver will be part of the kernel and the events will always proceed onto the driver and The impact of a fully functional system (kernel event logging + driver event copying + active trace daemon) is of 2.5% for core events. This means that for a task that took 100 seconds on a normal system, it will take 102.5 seconds on a traced system. This is very low compared to other profiling or tracing methods. For more Information about the Linux Trace Toolkit: `http://www.opersys.com/LTT/`

Chapter 18

Webresources

Resource	Description
Mainstream Kernel	
http://www.kernel.org	The offical site for the linux kernel.
Timers	
http://high-res-timers.sourceforge.net/	High Resolution Timer Project
http://www.cl.cam.ac.uk/ mgk25/time/c9/	Proposed new ¡time.h¿ for ISO C 200X
Low-Latency	
http://www.zipworld.com.au/ akpm/linux/	Andrew Mortons Low-Latency Patches
http://people.redhat.com/mingo/lowlatency-patches/	Ingo Molnars Low-Latency Patches
http://www.linuxdj.com/audio/lad/	Linux Audio Developers Mailing List
http://linux.oreillynet.com/pub/a/linux/2000/11/17/lowlatency.html	Article about low latency
http://www.linuxdj.com/audio/lad/resources.php3	Summary of Linux Audio Resources
	Benno Senoner has written some Latency test programs.
Preemtion Patch	
http://www.tech9.net/rml/linux	The preemption patch site - maintained by Robert Love
Benchmark Tools	
http://www.linuxdj.com/hdrbench/	high performance multitrack harddisk recording/playback benchmark
http://brain.mcmaster.ca/ hahn/realfeel.html	Realfeel - Realfeel Test of the Preemptible Kernel Patch

Chapter 19

Glossary

- APIC - On-chip interrupt controller provided on P6 and above Intel CPUs. Linux uses the timer interrupt register if a local APIC is available to provide its timer interrupt (is this true ?). The local APIC is part of a replacement for the old-style 8259 PIC, and receives external interrupts through an IO-APIC if there is

- ISR - Interrupt Service Routine, or interrupt handler. Also, on x86 APICs, In-Service Register, confusingly enough.

- DSR

- SMP

- pre-emption - Involuntary switching of a CPU from one task to another. User-space is pre-empted by interrupts, which can then either return to the process, or schedule another process (a process switch will also occur when the process "voluntarily" gives up the CPU by e.g. waiting for a disk block in kernel mode). Kernel mode tasks are never pre-empted (except by interrupts) - they are guaranteed use of the CPU until they sleep or yield the CPU. Some kernel code runs with interrupts disabled, meaning nothing except an NMI can interrupt the execution of the code.

- spinlock- A busy-wait method of ensuring mutual exclusion for a resource. Tasks waiting on a spin-lock sit in a busy loop until the spinlock becomes available. On a UP (single processor) system, spinlocks are not used and are optimised out of the kernel. There are also read-write spinlocks, where multiple readers are allowed, but only one writer. See Documentation/spinlocks.txt in the kernel source for details.

- Process/Thread/Task - The kernel abstraction for every user process, user thread, and kernel thread. All of these are handled as tasks in the kernel. Every task is described by its task_struct. User processes/threads have an associated user_struct. When in process context, the process's task_struct

is accessible through the routine get_current, which does assembly magic to access the struct, which is stored at the bottom of the kernel stack. When running in kernel mode without process context, the struct at the bottom of the kernel stack refers to the idle task. (taken from [?]

- ACPI - Advanced Configuration and Power Interface - replacement for APM that has the advantage of allowing O/S control of power management facilities.

- dynamic timers -

- Kernel Response Time - Is the time between the application request and the response from the kernel.

- Interactive -

t

Part III

Real Time Networking

Chapter 20

Introduction

Beginning with the introduction of RTOS, the issues of distributed real-time computing arose. A number of dedicated protocols and hardware concepts have been proposed and some realized, making well known RT-network protocols like CSMA/CD-NDBA or RTP and hardware implementations like CAN, Profi-bus, TTP, etc, inustry standards. Beyond the pure networking layer high level real-time distributed resource managment like RT-CORBA or MPI-RT [41] have been specified, but for their success in Linux based systems, a key issue seems to be providing real-time capabilities over inexpensive hardware. In part due to the history of Linux and in part due to the limited resources often available to Linux control "freaks" implementations focused on main-stream hardware, those being Serial lines (16550A UART), ethernet and firewire. Nevertheless as a response to the needs of the industry, CAN drivers for RTAI and RTLinux also have been developed.

In this part of the document we will focus on the real-time networking extensions available for real-time enhanced Linux systems. It should be noted though that due to the lack of reliable data for most real-time network implementations this document can't provide hard worst case timing information for most of the implementations (actually for none - as we did not have the resources to verify the published figures).

The document will present available real-time networking solutions on Linux platforms, their features and applications. For each of these solutions, following items will be evaluated:

1. Official Homepage (URL)
2. Licensing (GPL - which version, commercial...)
3. Availability of Source Code (yes/no)
4. Supported RTOS (Linux - which type, any other RTOS)

5. Supported Kernel Version (version number)
6. Starting Date of the Project
7. Latest Version (release date, version number)
8. Activity (low, high)
9. Number of Active Maintainers (their e-mail addresses if possible)
10. Supported HW Platforms
11. Supported Protocols
12. Supported I/O HW if applicable - (manufacturer, model)
13. Technical Support (available/not available, mailing list active/not active)
14. Applications (fields where this technology could be useful
15. Reference Projects (URLs, short description)
16. Performance (reported)
17. Documentation Quality
18. Contacts (e-mails of authors, maintainers...)

We will gather these information by extracting data from official and unofficial web sites of real-time networking solutions, by browsing mailing lists and by directly contacting authors and maintainers of these solutions.

At the time of writing, nine real-time networking solutions that can be implemented on Linux platforms, are known to us:

- RTcom
- spdrv
- RT-CAN
- RTnet
- lwIP for RTLinux
- LNET/RTLinuxPro Ethernet
- LNET/RTLinuxPro 1394 a/b
- RTsock
- TimeSys Linux/Net

Chapter 21

Real-Time Networking

If real-time performance of the network is to be achieved, two main sorts of problems need to be solved:

1. On the side of the network: network accessing policy
2. On the side of the RTOS: handling of messages through the interface-driver-OS-application path and back

21.1 Accessing the Network

The most popular and the most widely used network topology is a bus. In bus networks, access algorithms can be relatively simple, connecting and removing new units is easy and cabling is cheap. On a bus, all the nodes detect the transmitted message but each node decides autonomously if it should handle the message or drop it (one must not confuse this concept with broadcasting where messages are addressed to all the nodes in the network). Therefore, bus access policy, so called Arbitration, is the most important issue that needs to be solved by the bus designer. Different solutions exist and each has its advantages and disadvantages. Bus accessing policies decide about the complexity of implementation, about access delays, priority allocation, fairness of assigning bus-access, handling of faulty nodes etc.

In general, two types of arbitration exist:

- direct
- indirect

21.1.1 Direct Arbitration

A direct arbitration nowadays is mainly decentralized by delivering a so called "token" to a node which gets the right to send messages. After that, the token is delivered to another node, according to the implemented algorithm. The

concept is simple, but the implementation is quite demanding because all the error messages must be taken into consideration.

A special arbitration concept was developed for real-time networking. It is call Time-Slot mechanism. Each node in the network gets a guaranteed time-slot for delivering messages to the network. Assigning time-slots is static for the safety-critical applications, which makes it easy to guarantee that each node can really send and receive messages in its own time-slice. The term, used in telecommunications for time-slot concept, is a "synchronous bus-system". Each device in the network gets its own time-window through which it sends data, for example digitized voice. Arbitration in such a case is usually central. This mechanism is supported also by some widely used processors like MPC860. An example of this type of arbitration is IEEE 1394 FireWire with its isochronous transmission.

21.1.2 Indirect Arbitration

The indirect arbitration is widely spread in the LAN world where CSMA (Carrier Sense Multiple Access) contention protocol* is the most popular one. It is also used in field-bus world, especially in systems that require "soft" real-time behavior. This concept is also known as Random Access which implies that the devices can access the bus freely, whenever they want to although certain rules need to be defined beforehand. The most important one is that they need to test if some other device is active on the bus before accessing the bus by them selves. They are allowed to send messages only when the bus is free of any traffic. Though, this can lead to collisions that need to be resolved. Resolving collisions differentiates two types of CSMA contention protocol: CSMA/CD and CSMA/CA.

- CSMA/CD (Carrier Sense Multiple Access/ Collision Detection) contention protocol enables devices to detect a collision. The sending device is at the same time listening to the bus traffic and if it detects that its own signal has been damaged by some other sending device in the network, the sending device stops sending messages and waits for a certain delay time before trying again. The delay time after which it tries sending again is calculated by special algorithms, depending on the particular implementation. The best known protocol that uses CSMA/CD is Ethernet.

- CSMA/CA (Carrier Sense Multiple Access/ Collision Avoidance) listens to a network in order to avoid collisions, unlike CSMA/CD that deals with network transmissions once collisions have been detected. A device that is ready to send data, broadcasts a signal first in order to listen for collision scenarios and to tell other devices not to broadcast. This contributes to network traffic and lowers the useful network bandwidth. CSMA/CA is used by CAN protocol.

When real-time behavior is to be considered, no arbitration mechanism is ideal because the contradiction lies in the concept it self. Random access on one hand and a guaranteed transmitting performance on the other hand can only be achieved by a compromise. Different implementations, advertised on the market (even the Linux specific ones, described later on) are therefore more or less successful attempts to optimize the performance of a particular implementation. Though, this does raise a question if real-time networking is just a marketing buzz and even if it really needs to be implemented from the technological point of view in all the cases. There's no point in requiring a hard real-time performance of a network that consists of devices with a low level of reliability. In such cases, soft real-time performance is a more suitable requirement and it is also easier and cheaper to implement.

* A type of network protocol that allows nodes to contend for network access. That is, two or more nodes may try to send messages across the network simultaneously. The contention protocol defines what happens when this occurs. The most widely used contention protocol is CSMA/CD, used by Ethernet

21.2 RTOS Side of the Real-Time Networking

The main task that needs to be done by the OS to fulfill the real-time networking requirements is handling received and transmitted massages in a predictable time. The most obvious way to do that is to apply the handling mechanism on a real-time OS where resources for low latency, preemtive and predictable real-time performances are already available.

Networking extensions in real-time linux variants have appeard as early as the 2.2.2 kernel (tulip based RT-networking) and have since then extended into a number of different variants.

Although the focus here is on hard real-time networking many of these issues apply more or less unmodified to any form of soft real-time (QOS) networking as well. As there is resonably good documentation on the technologies utilized in the Linux networking layer [43] [42] [44] no summary of QOS and Differential Services in Linux is given; where Linux networking code is relevant to the rt-extension the involved specifics will be covered.

It should also be noted that there is an idirect relationship between Linux netwokring code an RT-performance as the networking subsystem puts a very high load on the memory subsystem of the Linux OS, thus optimizing the linux networking subsystem, or, in some cases, simply limiting its load, has a significan influence on the RT-performance [5].

Real-time networks are fundamentally different from non-rt, GPOS, networks. This section does not anticipate a complete coverage of the topic of real-time networking, but a few of the key technology issues should be noted and put into relation to the available implementations.

- Buffering
- Envelope Assembly/Disassembly
- Fragmentation
- Packet Interleaving/Dedicated Networks
- Error Handling
- Security
- Standardization

21.2.1 Buffering

A key problem of real-time networking implementations on the side of the RTOS has shown to be the strategy for the buffering of packets. As dynamic resource allocation is deprecated in hard real-time systems, different variants of prealocation have been developed. All implementations that anticipate hard real-time behavior have a fixed number of buffers prealocated and will swap pointers to these buffers during operation. Viewed from the system boundary moved to include all conected nodes this results in a double buffered strategy (receive and transmitt buffers must be prealocated).

Buffering strategies are fairly simple as long as one can assume non-fragmented communication. As this is not an acceptable limitation, buffering must also take the issue of fragmentation into account, which means that designing hard real-time network applications requires to take fragmentation into account and, if posible, prevent it, simplifying design and implementation a lot.

In implementations where buffering is not managed by the underlaying GPOS/RTOS subsystem, buffering needs to be included in the specifications for the application and appropriate test and validation needs to be concidered for the testing phase of a project, which means that if the buffering is not done by the underlaying OS then test and validation as well as specification efforts are increased and need to be taken into account.

21.2.2 Envelope Assembly/Disassembly

Although the problem of envelope assembly is fairly simple with respect to filling out the header(s), the real-time issue in this area is the necessary database queries. Hard real-time networking basically will mandate that all header related information is available at communication start, thus connection life-time is more or less equivalent to local task life-time. In princpal buffering , or caching, strategies are also possible with respect to protocol specific node informations, but to date no such strategies have been implemented in (any?) hard

real-time networking extensions to Linux.

CLEANUP:how is the packet header accessed

21.2.3 Fragmentation

Problems related to fragmentation were noted above in the paragraph on buffering, as most of the problems with fragmentation are related to buffering strategy. Asside from these there are also computational overhead for the housekeeping of fragmentation/defragmentation and the issue of increased latency in fragmenting systems, especially in those cases where the real-time networking layer also is available to the GPOS for packet transport. Furthermore there is a fragmentation related housekeeping overhead related to error handling if error cases should not be triggering a complete packet resend. It is also advisible to limit the data field of UDP datagram to some reasonable size to keep the network real-time.

21.2.4 Packet Interleaving/Dedicated Networks

If a single physical network connection on the wire level shall be used by the GPOS for non real-time packet delivery and at the same time guarantee real-time packet delivery with bounded latency, the strategy for interleaving of packets becomes a critical design issue. Generally the performance of interleving real-time/non real-time networks will be significantly lower than pure non real-time network links. The advantage being the reduced wireing and also the reduced system complexity, thus if performance permits shared networking then this is the prefered setup.
The alternative approach is simply to have dedicated links for GPOS and real-time trafic with independant media. Although this simplifies the analysis of the network transfer mechanism, there clearly is a danger of a highly loaded dedicated GPOS link impacting on the real-time network via the interrupt load generated, consequently dedicated networking setups, although simpler at first glance, may well be harder to predict in the real-time properties than shared network links with their increased internal complexity. To date not much work on this topic with relation to real-time enhanced networks for Linux has been done (TODO Phase2: Evaluate and benchmark shared vs. dedicated networks).

21.2.5 Error Handling

As with all real-time processing error haqndling is problematic, especially with respect to designing exit stratiges on fatal errors. The problem in the case of real-time networking is further agrevated due to the limited diagnostic posibilites of a node with respect to remote devicce status and recovery posibilities. To date all implementations leave error handling up to the application code in cases

where hard real-time communication is anticipated. De facto this means that the issue of error handling is not addressed in the available real-time networking imlementations.

21.2.6 Security

Generally security issues are simply neglected when it comes to real-time networking. Although fague security-initiatives have been announced, no project has addressed the issue of encryption protocols suited for real-time or the issue of DOS/DDOS in real-time networks, notably in such setups that utilize shared GPOS/RTOS trafic over the same physical link this seems problematic as there also is no simple strategy for external measures in such a setup (i.e. GPOS trafic behind firewals, RTOS trafic not conected to any non real-time nodes). The issue of security in real-time networks needs addressing if sensitive information transmition is anticipated (i.e. video-conferencing, VoIP conections etc.)

It should also be noted that the problem of authentication ("spoof protection") is currently not addressed in any of the realtieme networking implementations. Although this is typically a protocol issue, and thus seems inadequate here, it is noted here as any form of authentication would potentially introduce a communication overhead thus this issue needs addressing if hard real-time network capabilities are required. A posible solution seems to be to delegate authentication to non real-time processes and limit security of real-time transmission to encryption and compression.

21.2.7 Standardizatioan

Due to the inherent demands on real-time networking applications these are all de-facto non-standard APIs as soon as it comes to hard real-time networking , but for soft real-time implemenations POSIX complient APIs (socket layer) are evolving. It is not to be expected that the hard real-time networks will provide standard complient APIs due to the need for explicit buffer managment.

21.2.8 Open Issues

Issues that have not yet been addressed in the real-time networking extensions for real-time enhanced Linux are:

- Compression
- Encryption (data integrity)
- Authentication - Node Validation (spoof protection)
- Switching

- Complex Topologies

Building reliable and safe real-time networks, expecially when sharing the media with non real-time GPOS trafic, will be dependant on these issues being addressed in a suitable manner. At present none of the implementations seems to be addressing these issues and there are also no research projets known at this time that intend to include these topics (there are real-time Linux related projects and initiatives that pay attention to the security issue in general though (CLEANUP:ref orocos, ref FSMLabs security initiative))

21.2.9 CLEANUP:Hardware Related Issues

- linux-driver usage
- dedicated drivers
- stack implementations

21.2.10 CLEANUP:Non-RT Networking

- Remote system access and monitoring
- Monitoring of distributed systems
- non-RT clustering

Chapter 22

Notes on Protocols

This relatively large chapter covers protocol internals that we believe are necessary to be understood if different real-time networking implementations described later in the document are to be fairly evaluated. If the reader only wants to get an overview of the available real-time networking implementations or if she/he already posesses this knowledge, this chapter can be skipped. Otherwise it is strongly advised to read it through and get a good understanding of specifics of each of described protocols.

22.1 RS232/EIA232

RS-232 was created for one purpose, to interface between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE) employing serial binary data interchange. So as stated the DTE is the terminal or computer and the DCE is the modem or other communications device.

In the early 1960s, a standards committee, today known as the Electronic Industries Association, developed a common interface standard for data communications equipment. At that time, data communications was thought to mean digital data exchange between a centrally located mainframe computer and a remote computer terminal, or possibly between two terminals without a computer involved. These devices were linked by telephone voice lines, and consequently required a modem at each end for signal translation. While simple in concept, the many opportunities for data error that occur when transmitting data through an analog channel require a relatively complex design. It was thought that a standard was needed first to ensure reliable communication, and second to enable the interconnection of equipment produced by different manufacturers, thereby fostering the benefits of mass production and competition. From these ideas, the RS232 standard was born. It specified signal voltages, signal timing, signal function, a protocol for information exchange, and mechanical connectors.

Over the 40+ years since this standard was developed, the Electronic In-

dustries Association published three modifications, the most recent being the EIA232E standard introduced in 1991. Besides changing the name from RS232 to EIA232, some signal lines were renamed and various new ones were defined, including a shield conductor.

22.1.1 Serial Communications

The concept behind serial communications is as follows: data is transferred from sender to receiver one bit at a time through a single line or circuit. The serial port takes 8, 16 or 32 parallel bits from the computer bus and converts it as an 8, 16 or 32 bit serial stream. The name serial communications comes from this fact; each bit of information is transferred in series from one locations to another. In theory a serial link would only need two wires, a signal line and a ground, to move the serial signal from one location to another. But in practice this doesn't really work for a long time since some bits might get lost in the signal and thus alter the ending result. If one bit is missing at the receiving end, all succeeding bits are shifted resulting in incorrect data when converted back to a parallel signal. So to establish reliable serial communications one must overcome these bit errors that can emerge in many different forms.

Two serial transmission methods are used that correct serial bit errors. The first one is synchronous communication where the sending and receiving ends of the communication are synchronized using a clock that precisely times the period separating each bit. By checking the clock, the receiving end can determine if a bit is missing or if an extra bit (usually electrically induced) has been introduced in the stream. One important aspect of this method is that if either end of the communication loses it's clock signal, the communication is terminated.

The alternative method (used in PCs) is to add markers within the bit stream to help track each data bit. By introducing a start bit which indicates the start of a short data stream, the position of each bit can be determined by timing the bits at regular intervals. By sending start bits in front of each 8 bit streams, the two systems don't have to be synchronized by a clock signal, the only important issue is that both systems must be set at the same port speed. When the receiving end of the communication receives the start bit it starts a short term timer. By keeping streams short, there's not enough time for the timer to get out of sync. This method is known as asynchronous communication because the sending and receiving end of the communication are not precisely synchronized by the means of a signal line.

Each stream of bits is broken up in 5 to 8 bits called words. Usually in the PC environment you will find 7 or 8 bit words where the first is to accommodate all upper and lower case text characters in ASCII codes (the 127 characters) and the latter one is used to exactly correspond to one byte. By convention, the least significant bit of the word is sent first and the most significant bit is sent last. When communicating, the sender encodes the each word by adding a start bit in front and 1 or 2 stop bits at the end. Sometimes it will add a

parity bit between the last bit of the word and the first stop bit. This is used as a data integrity check and is often referred to as a data frame.

Five different parity bits can be used. The mark parity bit is always set at a logical 1, the space parity bit is always set at a logical 0, the even parity bit is set to logical 1 by counting the number of bits in the word and determining if the result is even. In the odd parity bit, the parity bit is set to logical 1 if the result is odd. The later two methods offer a means of detecting bit level transmission errors. Note that one doesn't have to use parity bits. Thus elliminating 1 bit in each frame, this is often reffered to as non parity bit frame.

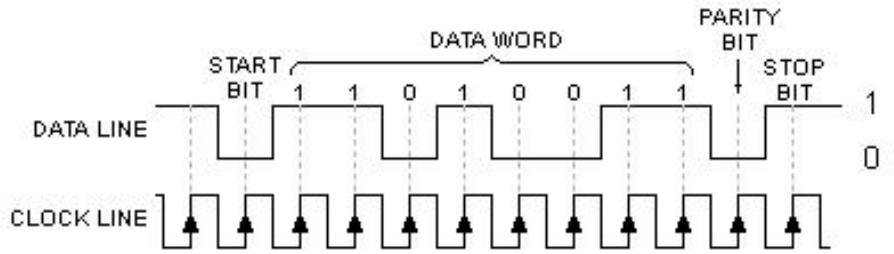

Figure 22.1: Asynchronous serial data frame (8E1)

In the Fig.22.1 you can see how the data frame is composed of and synchronised with the clock signal. This example uses an 8 bit word with even parity and 1 stop bit also refered to as an 8E1 setting.

22.1.2 Pin Assignments

Here is the full EIA232 signal definition for the DTE device (usually the PC). The most commonly used signals are shown in bold.

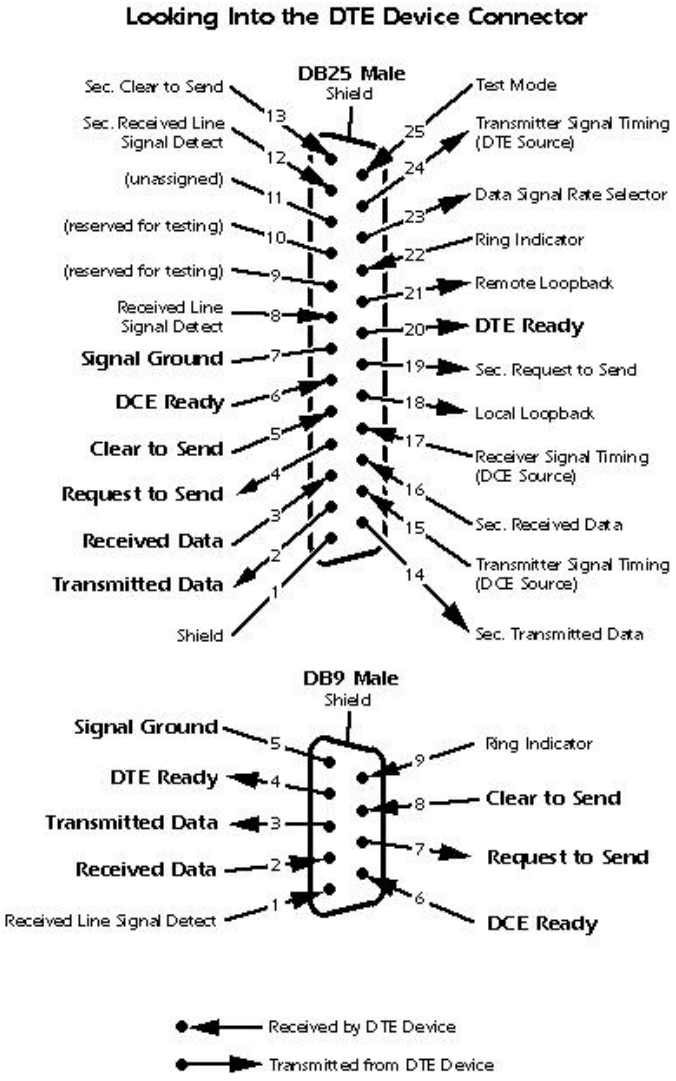

Figure 22.2: EIA232 signal definition for the DTE device

The Fig.22.3 shows the full EIA232 signal definition for the DCE device (usually the modem). The most commonly used signals are shown in bold.

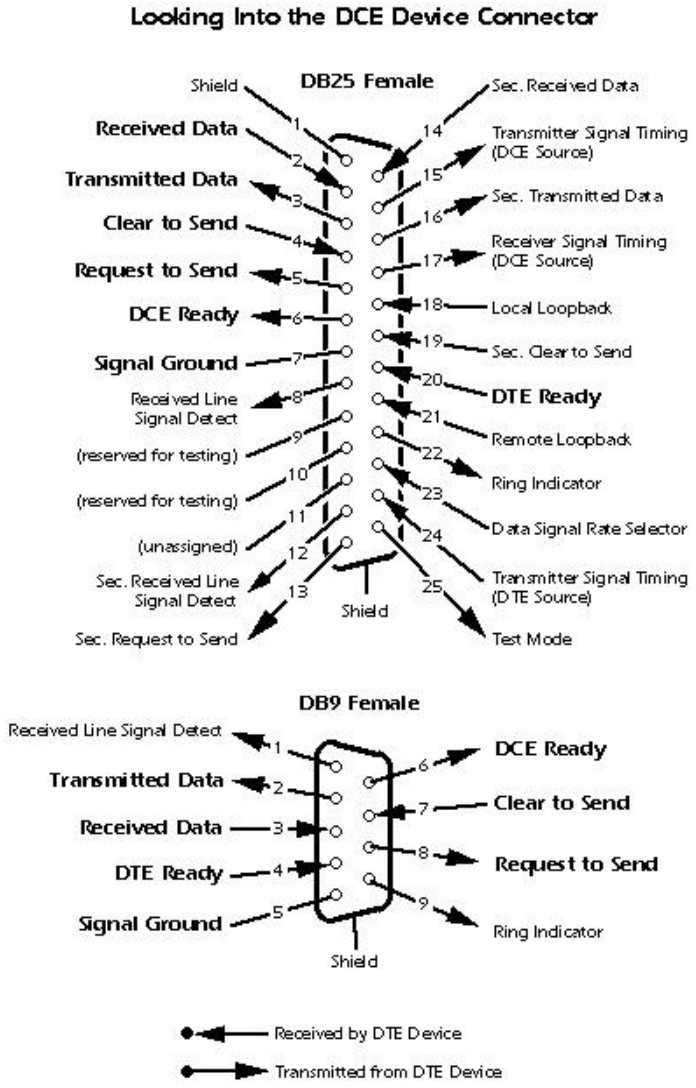

Figure 22.3: EIA232 signal definition for the DCE device

Signal names that imply a direction, such as Transmit Data and Receive Data, are named from the point of view of the DTE device. If the EIA232 standard were strictly followed, these signals would have the same name for the same pin number on the DCE side as well. Unfortunately, this is not done in practice by most engineers, probably because no one can keep straight which side is DTE and which is DCE. As a result, direction-sensitive signal names are changed at the DCE side to reflect their drive direction at DCE. The following list gives the conventional usage of signal names:

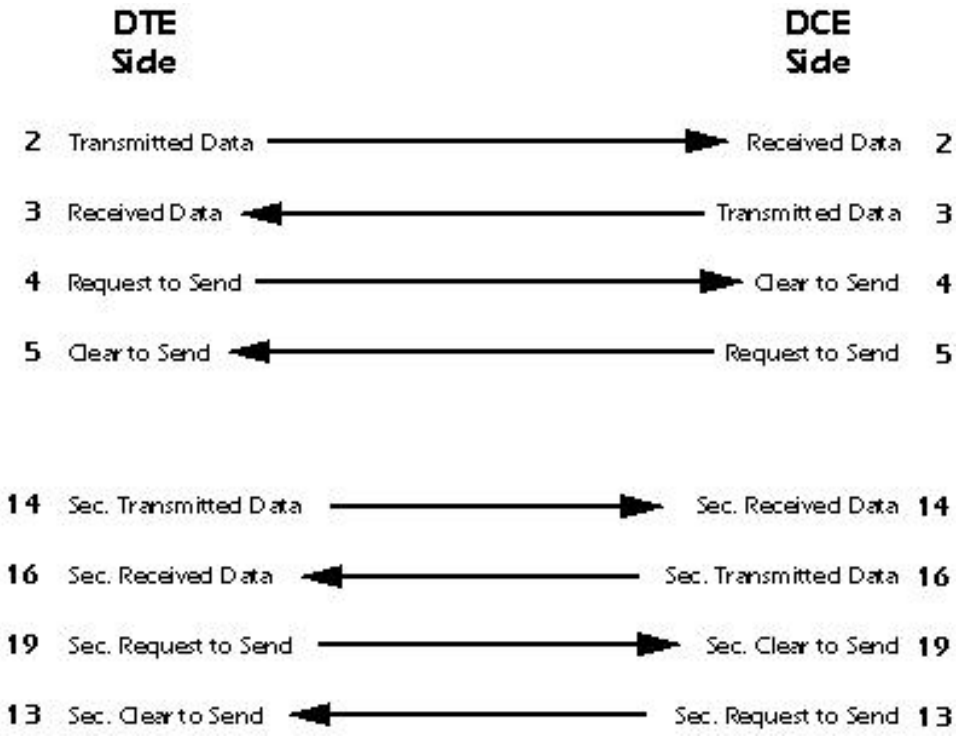

Figure 22.4: Conventional usage of signal names

22.2 CAN

Controlled Area Network (CAN) was introduced by Bosch in February 1986 at the Society of Automotive Engineers (SAE) congress and was primarily targeting the automotive market. Today, almost every new passenger car manufactured in Europe is equipped with at least one CAN network. Also used in other types of vehicles, from trains to ships, as well as in industrial controls, CAN is one of the most dominating bus protocols, maybe even the leading serial bus system worldwide. In 1999 alone, close to 60 million CAN controllers made their way into applications; more than 100 million CAN devices were sold in the year 2000.

The CAN protocol is an international standard defined in the ISO 11898. Beside the CAN protocol itself the conformance test for the CAN protocol is defined in the ISO 16845, which guarantees the interchangeability of the CAN chips.

Comparing to the most of the field-buses known at that time, CAN does not implement a node- but a message-oriented addressing. A message is characterized through an identifier that is 11 bit long in the standard frame and 29 bit long in the extended frame. Each node knows from his configuration which of these objects (messages) he is allowed to send and which he is allowed to receive. This makes upgrading of the CAN network much easier: the new

node doesn't have to know who he can communicate to but only needs to know which information is relevant for him presuming the assignment of identifiers to the messages is known in advance.

Message-oriented addressing implies that CAN is a multi-master, event-oriented system. Naturally, this must be followed by an appropriate bus-accessing (arbitration) mechanism. To avoid the usual delays, caused by collisions due to stochastic access to the bus, a CSMA/CA-NDBA (Carrier Sense Multiple Access with Collision Avoidance - Non Destructive Bit Arbitration) mechanism was chosen for CAN. Bus access conflicts are resolved by bit-wise arbitration on the identifiers involved by each station observing the bus level bit for bit. This happens in accordance with the "wired and" mechanism, by which the dominant state overwrites the recessive state. The competition for bus allocation is lost by all those stations (nodes) with recessive transmission and dominant observation. All those "losers" automatically become receivers of the message with the highest priority and do not re-attempt transmission until the bus is available again. Transmission requests are handled in the order of the importance of the messages for the system as a whole. This proves especially advantageous in overload situations. Since bus access is prioritized on the basis of the messages, it is possible to guarantee low individual latency times in real-time systems.

Bit-wise arbitration has also an important drawback: runtime of signals on the bus must be short comparing to the bit-time to enable a quasi concurrence for all the nodes in the network. The bus can be only 40 m long if a bit rate of 1 Mbit/s is to be achieved. This limitation is not that important in the automotive industry but it can lead to a reduced bit rate in the automation industry.

Unlike other bus systems, the CAN protocol does not use acknowledgement messages but instead signals any errors immediately as they occur. For error detection the CAN protocol implements three mechanisms at the message level: cyclic redundancy check (CRC), frame check and ACK errors.

The CAN protocol also implements two mechanisms for error detection at the bit level:

- monitoring
- bit stuffing

If one or more errors are discovered by at least one station using the above mechanisms, the current transmission is aborted by sending an "error flag". This prevents other stations accepting the message and thus ensures the consistency of data throughout the network. After transmission of an erroneous message that has been aborted, the sender automatically re-attempts transmission (automatic re-transmission). There may again competition for bus allocation. However effective and efficient the method described may be, in the event of a defective station it might lead to all messages (including correct ones) being aborted. If no measures fr self-monitoring were taken, the bus system would be

blocked by this. The CAN protocol therefore provides a mechanism to distinguishing sporadic errors from permanent errors and local failures at the station. This is done by statistical assessment of station error situations with the aim of recognizing a stations own defects and possibly entering an operation mode where the rest of the CAN network is not negatively affected. This may go as far as the station switching itself off to prevent messages erroneously from being recognized as incorrect .

CAN protocol it self defines only layers 1 and 2 of the ISO/OSI model. For exchanging short messages in a closed network this is sufficient. But applications in the industrial automation, higher-layer protocols are needed as well. CAN in Automation (CiA), the non-profit trade-association, has therefore defined a CAN Application Layer (CAL) and later on the CANopen protocol. CANopen is mainly used in machine control applications. In the factory automation, two more CAN-based protocols are mainly used: DeviceNet and Smart Distributed System (SDS). The protocol CAN Kingdom is used mostly in the safety-critical systems. These higher-layer protocols enable sending and receiving larger data segments and synchronization of nodes. Network management systems solve the problem of configuring nodes and assigning identifiers.

In the last few years an extension of the CAN protocol has also appeared. It is called Time Triggered CAN (TTCAN) and it is based on a periodic transmission of a reference message by a time master. This allows to introduce a system wide global network time with high precision. Based on this time the different messages are assigned to time windows within a basic cycle. A big advantage of TTCAN compared to classic scheduled systems is the possibility to transmit also event triggered messages in certain "arbitrating" time windows as well. These time windows, where normal arbitration takes place, allow the transmission of spontaneous messages. TTCAN is defined within ISO11898-4 standard.

22.3 IEEE 1394

IEEE 1394 was first introduced in the late 1980s by Apple Computer under the name FireWire. In the consumer electronics market it is more known as i.LINK.

The goal of the protocol is to provide easy-to-use, low-cost, high-speed communications. The protocol is also very scaleable, provides for both asynchronous and isochronous applications, allows for access to vast amounts of memory mapped address space, and - perhaps most important for the aforementioned convergence - allows peer-to-peer communication. Some people see 1394 and USB as competitors for the communications channel of the future, but in reality they are more complementary than competitive. USB is a lower-speed, lower-cost, host-based protocol and is suitable for lower-speed input devices such as keyboards, mice, joysticks, printers. IEEE 1394 is aimed at higher-speed multimedia peripherals such as video camcorders, set-top boxes, although slower speed devices like printers can also be connected to the IEEE 1394.

The only currently approved specification is the IEEE 1394-1995 specification, which was the basis for later extensions and enhancements. IEEE 1394-1995 supports transfer rates of 100, 200, and 400Mbps. As with many first cuts at a standard, 1394-1995 left some things up to the interpretation of the specifications implementers, which caused some interoperability problems and has led to the 1394a specification. This revision provides some clarification on the original specification, changes some optional portions of the spec to mandatory, and adds some performance enhancements. The 1394a specification was nearing completed in 2000. In addition to the 1394a specification, a 1394b specification was completed in 2002. 1394b provides for additional data rates of 800, 1,600, and 3,200Mbps. It also provides for long-haul transmissions via both twisted pair and fiber optics, and offers backward compatibility with the existing standard.

This section covers the 1394-1995 standard and will speak to some of the enhancements in the 1394a and 1394b revision.

22.3.1 Topology

The 1394 protocol is a peer-to-peer network with a point-to-point signaling environment. Nodes on the bus may have several ports on them. Each of these ports acts as a repeater, retransmitting any packets received by other ports within the node. Fig.22.5 shows what a typical consumer may have attached to their 1394 bus.

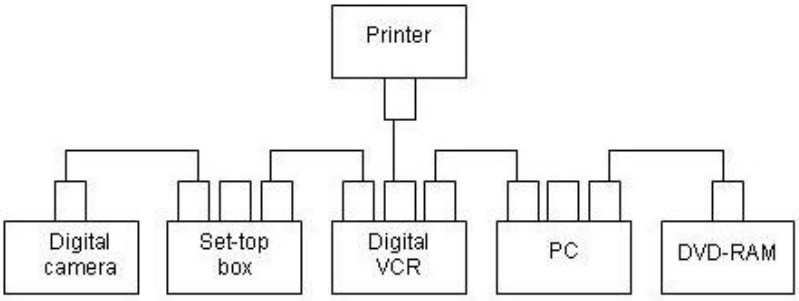

Figure 22.5: A firewire bus

Because 1394 is a peer-to-peer protocol, a specific host isnt required, such as the PC in USB. In the Fig.22.5, the digital camera could easily stream data to both the digital VCR and the DVD-RAM without any assistance from other devices on the bus.

Configuration of the bus occurs automatically whenever a new device is plugged in. Configuration proceeds from *leaf nodes* (those with only one other device attached to them) up through the branch nodes. A bus that has three or

more devices attached will typically, but not always, have a branch node become the root node.

A 1394 bus appears as a large memory-mapped space with each node occupying a certain address range. The memory space is based to the IEEE 1212 Control and Status Register (CSR) Architecture with some extensions specific to the 1394 standard. Each node supports up to 48 bits of address space (256 TeraBytes). In addition, each bus can support up to 64 nodes, and the 1394 serial bus specification supports up to 1,024 buses. This gives a grand total of 64 address bits, or support for a whopping total of 16 ExaBytes of memory space.

Transfers and Transactions

The 1394 protocol supports both asynchronous and isochronous data transfers, as will be presented in the following paragraphs.

Isochronous transfers. Isochronous transfers are always broadcast in a one-to-one or one-to-many fashion. No error correction and no retransmission are available for isochronous transfers. Up to 80% of the available bus bandwidth can be used for isochronous transfers. The delegation of bandwidth is tracked by a node on the bus that occupies the role of isochronous resource manager. This may or may not be the root node or the bus manager. The maximum amount of bandwidth an isochronous device can obtain is only limited by the number of other isochronous devices that have already obtained bandwidth from the isochronous resource manager.

Asynchronous transfers. Asynchronous transfers are targeted to a specific node with an explicit address. They are not guaranteed a specific amount of bandwidth on the bus, but they are guaranteed a fair shot at gaining access to the bus when asynchronous transfers are permitted.

The maximum data block size for an asynchronous and for an isochronous packet is determined by the transfer rate of the device, as specified in Table 22.1.

Speed	Max data payload (in bytes)	
	Asynchronous	Isochronous
100 Mb/s	512	1024
200 Mb/s	1024	2048
400 Mb/s	2048	4096

Table 22.1: Minimum data block size

Asynchronous transfers are acknowledged and responded to. This allows error-checking and retransmission mechanisms to take place.

The bottom line is that if you're sending time-critical, error-tolerant data, such as a video or audio stream, isochronous transfers are the way to go. If the data isn't error-tolerant, such as a disk drive, then asynchronous transfers are preferable. The 1394 specification defines four protocol layers, known as the physical layer, the link layer, the transaction layer, and the serial bus management layer. The layers are illustrated in Fig.22.6.

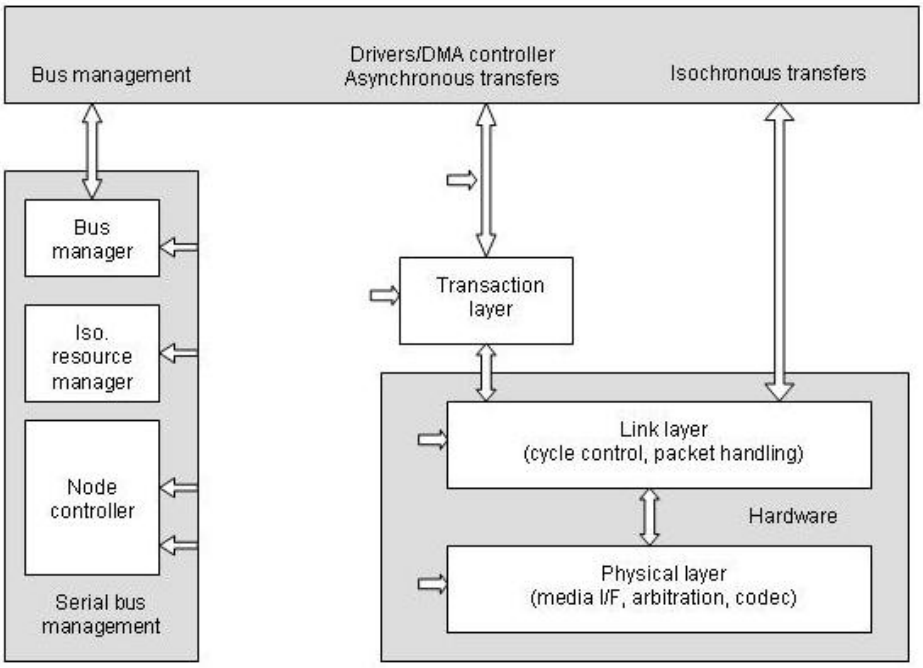

Figure 22.6: IEEE-1394 protocol layers

22.3.2 Physical layer

The physical layer of the 1394 protocol includes the electrical signaling, the mechanical connectors and cabling, the arbitration mechanisms, and the serial coding and decoding of the data being transferred or received. The cable media is defined as a three-pair shielded cable. Two of the pairs are used to transfer data, while the third pair provides power on the bus. The connectors are small six-pin devices, although the 1394a also defines a four-pin connector for self-powered leaf nodes. The power signals arent provided on the four-pin connector. The baseline cables are limited to 4.5m in length. Thicker cables allow for longer

distances.

The two twisted pairs used for signaling, called out as TPA and TPB, are bidirectional and are tri-state capable. TPA is used to transmit the strobe signal and receive data, while TPB is used to receive the strobe signal and transmit data. The signaling mechanism uses data strobe encoding, a rather clever technique that allows easy extraction of a clock signal with much better jitter tolerance than a standard clock/data mechanism. With data strobe encoding, either the data or the strobe signal (but not both of them) change in a bit cell. Data strobe encoding is shown in Fig.22.7.

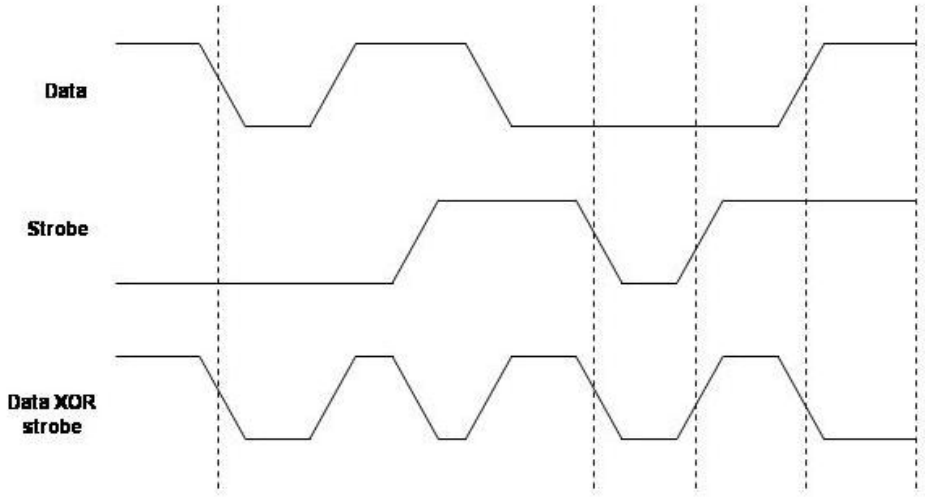

Figure 22.7: Data strobe encoding

Configuration

The physical layer plays a major role in the bus configuration and normal arbitration phases of the protocol. Configuration consists of taking a relatively flat physical topology and turning it into a logical tree structure with a root node at its focal point. A bus is reset and reconfigured whenever a device is added or removed. A reset can also be initiated via software. Configuration consists of bus reset and initialization, tree identification, and self identification.

Reset. Reset is signaled by a node driving both TPA and TPB to logic 1. Because of the "dominant 1s" electrical definition of the drivers, a logic 1 will always be detected by a port, even if its bidirectional driver is in the transmit state. When a node detects a reset condition on its drivers, it will propagate this signal to all of the other ports that this node supports. The node then enters the idle state for a given period of time to allow the reset indication to propagate to all other nodes on the bus. Reset clears any topology information within the node, although isochronous resources are "sticky" and will tend to

remain the same during resets.

Tree identification. The tree identification process defines the bus topology. Let's take the example of our sample home consumer network. After reset, but before tree identification, the bus has a flat logical topology that maps directly to the physical topology. After tree identification is complete, a single node has gained the status of root node. The tree identification proceeds as follows.

After reset, all leaf nodes present a Parent_Notify signaling state on their data and strobe pairs. Note that this is a signaling state, not a transmitted packet. The whole tree identification process occurs in a matter of microseconds. In our example, the digital camera will signal the set-top box, the printer will signal the digital VCR, and the DVD-RAM will signal the PC. When a branch node receives the Parent_Notify signal on one of its ports, it marks that port as containing a child, and outputs a Child_Notify signaling state on that port's data and strobe pairs. Upon detecting this state, the leaf node marks its port as a parent port and removes the signaling, thereby confirming that the leaf node has accepted the *child* designation. At this point our bus appears as shown in Fig.22.8

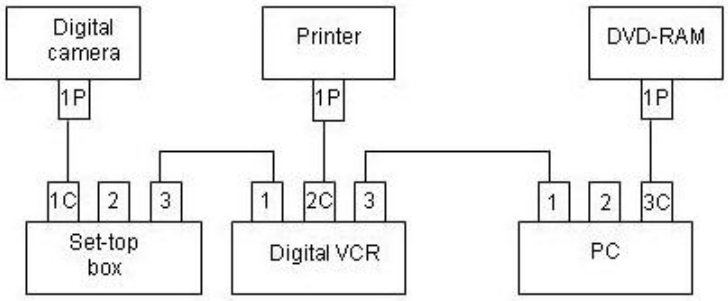

Figure 22.8: Bus after leaf node identification

The ports marked with a "P" indicate that a device which is closer to the root node is attached to that port, while a port marked with a "C" indicates that a node farther away from the root node is attached. The port numbers are arbitrarily assigned during design of the device and play an important part in the self identification process.

After the leaf nodes have identified themselves, the digital VCR still has two ports that have not received a Parent_Notify, while the set-top box and the PC branch node both have only one port with an attached device that has not received a Parent_Notify. Therefore, both the set-top box and the PC start to signal a Parent_Notify on the one port that has not yet received one. In this case, the VCR receives the Parent_Notify on both of its remaining ports, which it acknowledges with a Child_Notify condition. Because the VCR has marked all

of its ports as children, the VCR becomes the root node. The final configuration is shown in Fig.22.9

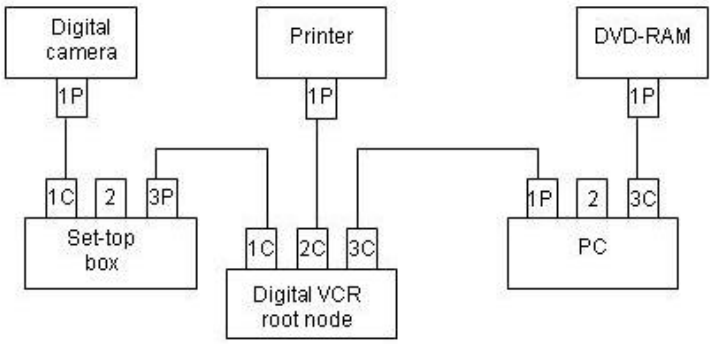

Figure 22.9: Bus after tree identification is complete

Note that two nodes can be in contention for root node status at the end of the process. In this case, a random back-off timer is used to eventually settle on a root node. A node can also force itself to become root node by delaying its participation in the tree identification process for a while.

Self identification. Once the tree topology is defined, the self identification phase begins. Self identification consists of assigning physical IDs to each node on the bus, having neighboring nodes exchange transmission speed capabilities, and making all of the nodes on the bus aware of the topology that exists. The self identification phase begins with the root node sending an arbitration grant signal to its lowest numbered port. In our example, the digital VCR is the root node and it signals the set-top box. Since the set-top box is a branch node, it will propagate the Arbitration Grant signal to its lowest numbered port with a child node attached. In our case, this port is the digital camera. Because the digital camera is a leaf node, it cannot propagate the arbitration grant signal downstream any farther, so it assigns itself physical ID 0 and transmits a self ID packet upstream. The branch node (set-top box) repeats the self ID packet to all of its ports with attached devices. Eventually the self ID packet makes its way back up to the root node, which proceeds to transmit the self ID packet down to all devices on its higher-numbered ports. In this manner, all attached devices receive the self ID packet that was transmitted by the digital camera. Upon receiving this packet, all of the other devices increment their self ID counter. The digital camera then signals a self ID done indication upstream to the set-top box, which indicates that all nodes attached downstream on this port have gone through the self ID process. Note that the set-top box does not propagate this signal upstream toward the root node because it hasn't completed the self ID process.

The root node will then continue to signal an Arbitration Grant signal to its lowest numbered port which in this case is still the set-top box. Because the set-top box has no other attached devices, it assigns itself physical ID 1 and transmits a self ID packet back upstream. This process continues until all ports on the root node have indicated a *self ID done* condition. The root node then assigns itself the next physical ID. The root node will always be the highest-numbered device on the bus. If we follow through with our example, we come up with the following physical IDs: digital camera = 0; set-top box = 1; printer = 2; DVD-RAM = 3; PC = 4; and the digital VCR, which is the root node, 5.

Note that during the self ID process, parent and children nodes are also exchanging their maximum speed capabilities. This process also exposes the Achilles heel of the 1394 protocol. Nodes can only transmit as fast as the slowest device between the transmitting node and the receiving node. For example, if the digital camera and the digital VCR are both capable of transmitting at 400Mbps, but the set-top box is only capable of transmitting at 100Mbps, the high-speed devices cannot use the maximum rate to communicate amongst themselves. The only way around this problem is for the end user to reconfigure the cabling so the low-speed set-top box is not physically between the two high-speed devices.

Also during the self ID process, all nodes wishing to become the isochronous resource manager will indicate this fact in their self ID packet. The highest numbered node that wishes to become resource manager will receive the honor.

Normal Arbitration

Once the configuration process is complete, normal bus operations can begin. To fully understand arbitration, a knowledge of the cycle structure of 1394 is necessary.

A 1394 cycle is a time slice with a nominal 125s period. The 8kHz cycle clock is kept by the cycle master, which is also the root node. To begin a cycle, the cycle master broadcasts a cycle start packet, which all other devices on the bus use to synchronize their timebases.

Immediately following the cycle start packet, devices that wish to broadcast their isochronous data may arbitrate for the bus. Arbitration consists of signaling your parent node that you wish to gain access to the bus. The parent nodes in turn signal their parents and so on, until the request reaches the root node. In our previous example, suppose the digital camera and the PC wish to stream data over the bus. They both signal their parents that they wish to gain access to the bus. Since the PC's parent is the root node, its request is received first and it is granted the bus. From this scenario, it is evident that the closest device to the root node wins the arbitration.

Because isochronous channels can only be used once per cycle, when the next isochronous gap occurs, the PC will no longer participate in the arbitration.

This condition allows the digital camera to win the next arbitration. Note that the PC could have more than one isochronous channel, in which case it would win the arbitration until it had no more channels left. This points out the important role of the isochronous resource manager: it will not allow the allotted isochronous channels to require more bandwidth than available.

When the last isochronous channel has transmitted its data, the bus becomes idle waiting for another isochronous channel to begin arbitration. Because there are no more isochronous devices left waiting to transmit, the idle time extends longer than the isochronous gap until it reaches the duration defined as the subaction (or asynchronous) gap. At this time, asynchronous devices may begin to arbitrate for the bus. Arbitration proceeds in the same manner, with the closest device to the root node winning arbitration.

This point brings up an interesting scenario: because asynchronous devices can send more than one packet per cycle, the device closest to the root node (or the root node itself) might be able to hog the bus by always winning the arbitration. This scenario is dealt with using what is called the *fairness interval* and the arbitration rest gap. The concept is simpleonce a node wins the asynchronous arbitration and delivers its packet, it clears its arbitration enable bit. When this bit is cleared, the physical layer no longer participates in the arbitration process, giving devices farther away from the root node a fair shot at gaining access to the bus. When all devices wishing to gain access to the bus have had their fair shot, they all wind up having their arbitration enable bits cleared, meaning no one is trying to gain access to the bus. This causes the idle time on the bus to go longer than the 10s subaction gap until it finally reaches 20s, which is called the arbitration reset gap. When the idle time reaches this point, all devices may reset their arbitration enable bits and arbitration can begin all over again.

22.3.3 Link Layer

The link layer is the interface between the physical layer and the transaction layer. The link layer is responsible for checking received CRCs and calculating and appending the CRC to transmitted packets. In addition, because isochronous transfers do not use the transaction layer, the link layer is directly responsible for sending and receiving isochronous data. The link layer also examines the packet header information and determines the type of transaction that is in progress. This information is then passed up to the transaction layer.

The interface between the link layer and the physical layer is listed as an informative (not required) appendix in the IEEE 1394-1995 specification. In the 1394a addendum, however, this interface becomes a required part of the specification. This change was instituted to promote interoperability amongst the various 1394 chip vendors.

The link layer to physical layer interface consists of a minimum of 17 signals that must be either magnetically or capacitively isolated from the PHY. These signals are defined in Table 22.2.

Signal	Source	Description
LReq	Link request	Link request—used to initiate a request to send a packet, as well as a request to read directly from a PHY register.
SClk	Physical layer	49.152MHz clock used to synchronize data readout. (The frequency may change depending on data rates with 1394b.)
Data[0:7]	Either	Data—higher transfer speeds use an increasing number of bits: 100Mbps —D[0:1] 200Mbps —D[0:3] 400Mbps —D[0:7] Note that the width of this data bus may expand to 16 bits with 1394b.
Ctl[0:1]	Either	Control interface—defines what state the interface is in.
LPS	Link layer	Link power status—indicates that the link layer controller is powered.
Link On	Physical layer	Indicates that the physical layer has been powered on.
Direct	Neither	Indicates that no isolation barrier exists.
Backplane	Physical layer	High if physical layer is a backplane implementation.
Clk	Neither	Indicates that SClk is only 24.576MHz; valid in a backplane implementation only.

Table 22.2: Seventeen signals of the link layer to physical layer interface

A typical link layer implementation has the PHY interface, a CRC checking and generation mechanism, transmit and receive FIFOs, interrupt registers, a host interface and at least one DMA channel.

22.3.4 Transaction Layer

The transaction layer is used for asynchronous transactions. The 1394 protocol uses a request-response mechanism, with confirmations typically generated within each phase. Several types of transactions are allowed. They are listed as follows:
Simple quadlet (four-byte) read
Simple quadlet write
Variable-length read
Variable-length write
Lock transactions

Lock transactions allow for atomic swap and compare and swap operations to be performed. Asynchronous packets have a standard header format, along with an optional data block. The packets are assembled and disassembled by the link layer controller. Fig.22.10 shows the format of a typical asynchronous packet.

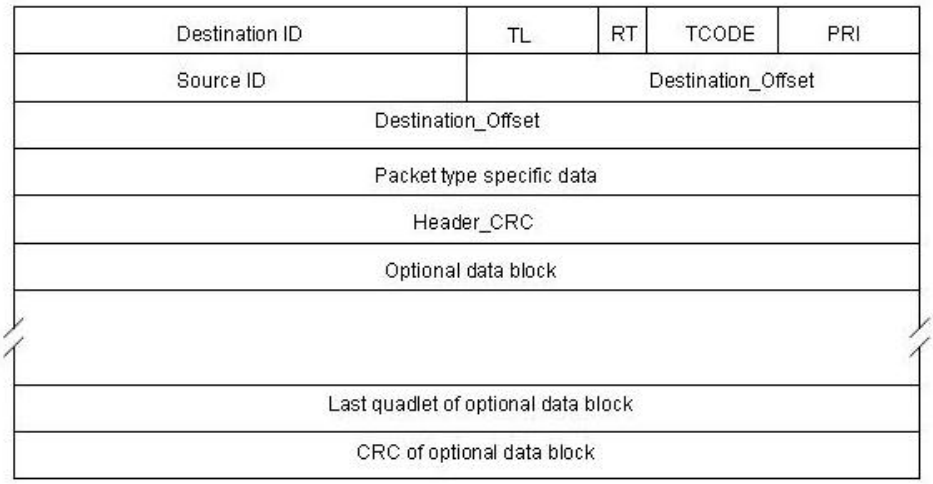

Figure 22.10: Asynchronous packet format

Transactions can be split, concatenated, or unified. Fig.22.11 illustrates a split transaction. The split transaction occurs when a device cannot respond fast enough to the transaction request. When a request is received, the node responds with an acknowledge packet. An acknowledge packet is sent after every asynchronous packet. In fact, the acknowledging device doesn't even have to arbitrate for the bus; control of the bus is automatic after receiving an incoming request or response packet.

As you can see, the responder node sends the acknowledge back and then prepares the data that was requested. While this is going on, other devices may be using the bus. Once the responder node has the data ready, it begins to arbitrate for the bus, to send out its response packet containing the desired data. The requester node receives this data and returns an acknowledge packet (also without needing to re-arbitrate for the bus).

If the responder node can prepare the requested data quickly enough, the entire transaction can be concatenated. This removes the need for the responding node to arbitrate for the bus after the acknowledge packet is sent.

For data writes, the acknowledgement can also be the response to the write, which is the case in a unified transaction. If the responder can accept the data fast enough, its acknowledge packet can have a transaction code of *complete* instead of *pending*. This eliminates the need for a separate response transaction

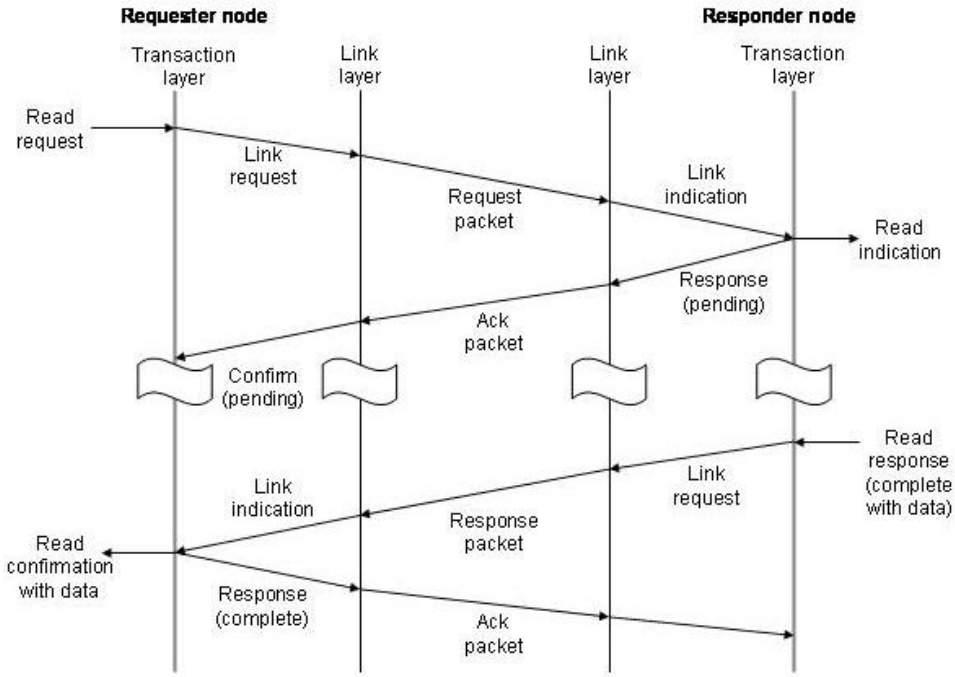

Figure 22.11: A split transaction

altogether. Note that unified read and lock transactions aren't possible, and the acknowledge packet can't return data.

1394a Arbitration Enhancements

The 1394a addendum adds three new types of arbitration to be used with asynchronous nodes: acknowledged accelerated arbitration, fly-by arbitration, and token-style arbitration.

Acknowledged accelerated arbitration. When a responding node also has a request packet to transmit, the responding node can immediately transmit its request without arbitrating for the bus. Normally the responding node would have to go through the standard arbitration process.

Fly-by arbitration. A node that contains several ports must act as a repeater on its active ports. A multiport node may use fly-by arbitration on packets that dont require acknowledgement (isochronous packets and acknowledge packets). When a node using this technique is repeating a packet *upstream* toward the root node, it may concatenate an identical speed packet to the end of the current packet. Note that asynchronous packets may not be added to isochronous packets.

Token-style arbitration. Token-style arbitration requires a group of cooperating nodes. When the cooperating node closest to the root node wins a normal

arbitration, it can pass the arbitration grant down to the node farthest from the root. This node sends a normal packet, and all of the cooperating nodes can use fly-by arbitration to add their packets to the original packet as it heads upstream.

22.3.5 Bus Management Layer

Bus management on a 1394 bus involves several different responsibilities that may be distributed among more than one node. Nodes on the bus must assume the roles of cycle master, isochronous resource manager, and bus manager.

Cycle master. The cycle master initiates the 125s cycles. The root node must be the cycle master; if a node that is not cycle master capable becomes root node, the bus is reset and a node that is cycle master capable is forced to be the root. The cycle master broadcasts a cycle start packet every 125s. Note that a cycle start can be delayed while an asynchronous packet is being transmitted or acknowledged. The cycle master deals with this by including the amount of time that the cycle was delayed in the cycle start packet.

Isochronous resource manager. The isochronous resource manager must be isochronous transaction capable. The isochronous resource manager must also implement several additional registers. These registers include the Bus Manager ID Register, the Bus Bandwidth Allocation Register, and the Channel Allocation Register. Isochronous channel allocation is performed by a node that wishes to transmit isochronous packets. These nodes must allocate a channel from the Channel Allocation Register by reading the bits in the 64-bit register. Each channel has one bit associated with it. A channel is available if its bit is set to a logic 1. The requesting node sets the first available channel bit to a logic 0 and uses this bit number as the channel ID.

In addition, the requesting node must examine the Bandwidth Available Register to determine how much bandwidth it can consume. The total amount of bandwidth available is 6,144 allocation units. One allocation unit is the time required to transfer one quadlet at 1,600Mbps. A total of 4,915 allocation units are available for isochronous transfers if any asynchronous transfers are used. Nodes wishing to use isochronous bandwidth must subtract the amount of bandwidth needed from the Bandwidth Available Register.

Bus manager. A bus manager has several functions, including publishing the topology and speed maps, managing power, and optimizing bus traffic. The topology map may be used by nodes with a sophisticated user interface that could instruct the end user on the optimum connection topology to enable the highest throughput between nodes. The speed map is used by nodes to determine what speed it can use to communicate with other nodes.

The bus manager is also responsible for determining whether the node that has become root node is cycle master capable. If it isn't, the bus manager searches for a node that is cycle master capable and forces a bus reset that will select that node as root node. The bus manager might not always find a

capable node; in this case, at least some of the bus management functions are performed by the isochronous resource manager.

22.3.6 1394b

An enhanced specification, 1394b was finalized in 2002. IEEE 1394b extends bus speeds to 800 and 1,600 Mbps. The enhancements also include architectural support for 3,200 Mbps, although the signaling parameters for 3,200 Mbps are not yet available. The IEEE 1394b also supports forms of cabling not supported in the existing 1394a specification, resulting in a dramatic increase in cable lengthsfrom the 4.5 meters of the original standard copper cable to 100 meters for plastic optical fiber, multiple kilometers (km) for glass optical fiber cables, and 100 meters for category 5 (CAT-5) at 100 Mbps.

22.4 Ethernet

IEEE has produced several standards for LANs. These standards, collectively known as IEEE 802, include CSMA/CD, token bus and token ring. The various standards differ at the physical layer and Media Access Control (MAC) sublayer but are compatible at the data link layer.

The standards are divided into parts, each published as a separate book. The 802.1 standard gives an introduction to the set of standards and defines the interface primitives. The 802.2 standard describes the upper part of the data link layer, which uses the LLC (Logical Link Control) protocol. Parts 802.3 through 802.5 describe the three LAN standards, the CSMA/CD, token bus, and token ring standards, respectively. Each standard covers the physical layer and MAC sublayer protocol.

The IEEE 802.3 standard is for a 1-persistent CSMA/CD LAN. To review the idea, when a station wants to transmit, it listens to the cable. If the cable is busy, the station waits until it goes idle; otherwise it transmits immediately. If two or more stations simultaneously begin transmitting on an idle cable, they will collide. All colliding stations then terminate their transmission, wait a random time, and repeat the whole process all over again. The protocol is called 1-persistent because the station transmits with a probability of 1 whenever it finds the channel idle (also p-persistent and non-persistent CSMA/CD protocols exist, but they are not part of IEEE 802.3 standard).

The term *Ethernet* refers to the family of local-area network (LAN) products covered by the IEEE 802.3 standard although many people (incorrectly) use the name "Ethernet" in a generic sense to refer to all CSMA/CD protocols.

The original Ethernet was developed as an experimental coaxial cable network in the 1970s by Xerox to operate with a data rate of 3 Mbps using a CSMA/CD protocol for LANs with sporadic but occasionally heavy traffic requirements. This system was called Ethernet after the *luminiferous ether*,

through which electromagnetic radiation was once thought to propagate. Success of the project attracted early attention and led to the 1980 joint development of the 10-Mbps Ethernet Version 1.0 specification by the three-company consortium: Digital Equipment Corporation, Intel, and Xerox. This specification formed the basis for 802.3. The published 802.3 standard differs from the Ethernet specification in that it describes a whole family of 1-peristent CSMA/CD systems, running at speeds from 1 to 10-Mbps on various media. Also, the one header field differs between the two (the 802.3 length field is used for packet type in Ethernet). The initial standard also gives the parameters for a 10 Mbps base-band system using 50-ohm coaxial cable. Parameter sets for other media and speeds came later.

Four data-rates are currently defined for operation over optical fiber and twisted-pair cables:

10 Mbps - 10Base-T Ethernet
100 Mbps - Fast Ethernet
1000 Mbps - Gigabit Ethernet
10 Gbps - 10 Gigabit Ethernet

The IEEE 802.3 standard currently requires that all the Ethernet MACs support half-duplex operation, in which the MAC can be either transmitting or receiving a frame, but it cannot be doing both simultaneously. Full-duplex operation is an optional MAC capability that allows the MAC to transmit and receive frames simultaneously.

22.4.1 Ethernet Network Elements

Ethernet LANs consist of network nodes and interconnecting media. The network nodes fall into two major classes:

Data terminal equipment (DTE) - Devices that are either the source or the destination of data frames. DTEs are typically devices such as PCs, workstations, file servers, or print servers that, as a group, are all often referred to as end stations.

Data communication equipment (DCE) - Intermediate network devices that receive and forward frames across the network. DCEs may be either standalone devices such as repeaters, network switches, and routers, or communications interface units such as interface cards and modems.

Throughout this section, standalone intermediate network devices will be referred to as either *intermediate nodes* or DCEs. Network interface cards will be referred to as *NICs*. The current Ethernet media options include two general types of copper cable: unshielded twisted-pair (UTP) and shielded twisted-pair (STP), plus several types of optical fiber cable.

22.4.2 The IEEE 802.3 Logical Relationship to the ISO Reference Model

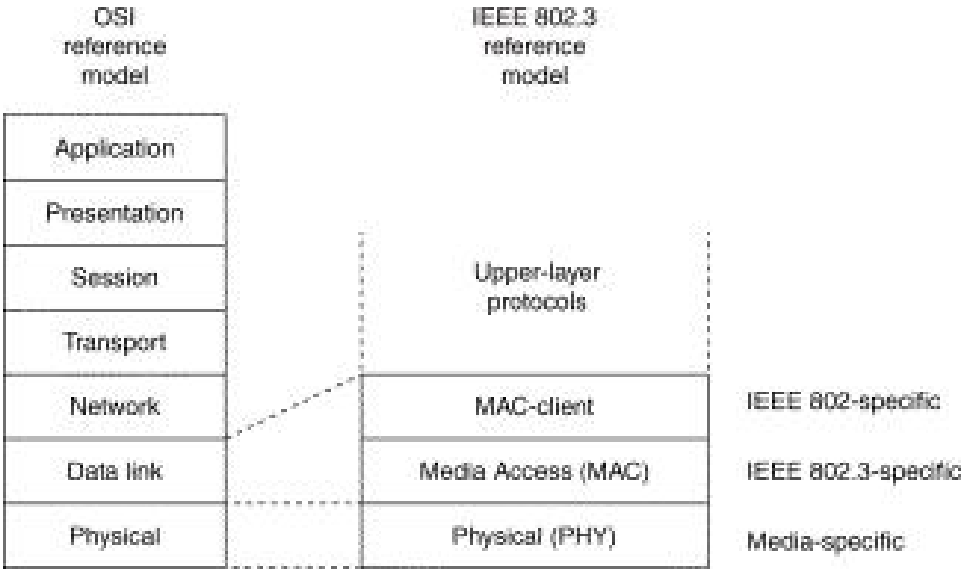

Figure 22.12: Ethernet's logical relationship to the ISO reference model

The MAC-client sublayer may be one of the following:

Logical Link Control (LLC), if the unit is a DTE. This sublayer provides the interface between the Ethernet MAC and the upper layers in the protocol stack of the end station. The LLC sublayer is defined by IEEE 802.2 standards.

Bridge entity, if the unit is a DCE. Bridge entities provide LAN-to-LAN interfaces between LANs that use the same protocol (for example, Ethernet to Ethernet) and also between different protocols (for example, Ethernet to Token Ring). Bridge entities are defined by IEEE 802.1 standards.

Because specifications for LLC and bridge entities are common for all IEEE 802 LAN protocols, network compatibility becomes the primary responsibility of the particular network protocol.

The MAC layer controls the node's access to the network media and is specific to the individual protocol. All IEEE 802.3 MACs must meet the same basic set of logical requirements, regardless of whether they include one or more of the defined optional protocol extensions. The only requirement for basic communication (communication that does not require optional protocol extensions) between two network nodes is that both MACs must support the same transmission rate. The 802.3 physical layer is specific to the transmission data rate, the signal encoding, and the type of media interconnecting the two nodes. Gigabit Ethernet, for example, is defined to operate over either twisted-pair or optical fiber cable, but each specific type of cable or signal-encoding procedure requires a different physical layer implementation.

22.4.3 Network Topologies

LANs take on many topological configurations, but regardless of their size or complexity, all will be a combination of only three basic interconnection structures or network building blocks. The simplest structure is the point-to-point interconnection. Only two network units are involved, and the connection may be DTE-to-DTE, DTE-to-DCE, or DCE-to-DCE. The cable in point-to-point interconnections is known as a network link. The maximum allowable length of the link depends on the type of cable and the transmission method that is used. The original Ethernet networks were implemented with a coaxial bus structure. Segment lengths were limited to 500 meters, and up to 100 stations could be connected to a single segment. Individual segments could be interconnected with repeaters, as long as multiple paths did not exist between any two stations on the network and the number of DTEs did not exceed 1024. The total path distance between the most-distant pair of stations was also not allowed to exceed a maximum prescribed value. Although new networks are no longer connected in a bus configuration, some older bus-connected networks do still exist and are still useful. Since the early 1990s, the network configuration of choice has been the star-connected topology. The central network unit is either a multiport repeater (also known as a hub) or a network switch. All connections in a star network are point-to-point links implemented with either twisted-pair or optical fiber cable.

22.4.4 Manchester Encoding

None of the versions of 802.3 use straight binary encoding with 0 volts for a 0 bit and 5 volts for a 1 bit because it leads to ambiguities. If one station sends the bit string 0001000, others might falsely interpret it as 10000000 or 01000000 because they cannot tell the difference between an idle sender (0 volts) and a 0 bit (0 volts).

What is needed is a way for receivers to unambiguously determine the start, end, or middle of each bit without reference to an external clock. Such an approach is called *Manchester encoding*. With Manchester encoding, each bit period is divided into two equal intervals. A binary 1 bit is sent by having the voltage set high during the first interval and low in the second one. A binary 0 is just the reverse: first low and then high. This scheme ensures that every bit period has a transition in the middle, making it easy for the receiver to synchronize with the sender.

A disadvantage of Manchester encoding is that it requires twice as much bandwidth as straight binary encoding, because the pulses are half the width. This makes it unsuitable for use at higher data rates and Ethernet versions subsequent to 10Base-T all use different encoding procedures that include some or all of the following techniques:

Using data scrambling - A procedure that scrambles the bits in each byte

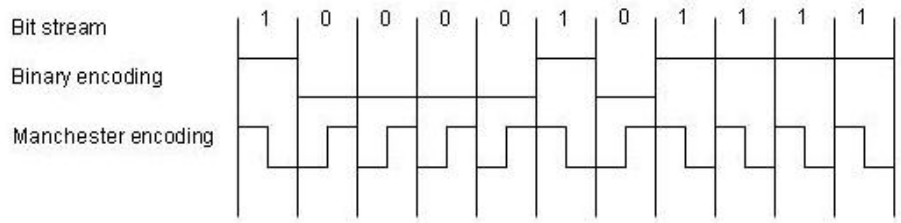

Figure 22.13: (a) Binary encoding (b) Manchester encoding

in an orderly (and recoverable) manner. Some 0s are changed to 1s, some 1s are changed to 0s, and some bits are left the same. The result is reduced run-length of same-value bits, increased transition density, and easier clock recovery.

Expanding the code space - A technique that allows assignment of separate codes for data and control symbols (such as start-of-stream delimiters, extension bits, and so on) and that assists in transmission error detection.

Using forward error-correcting codes - An encoding in which redundant information is added to the transmitted data stream so that some types of transmission errors can be corrected during frame reception.

22.4.5 The 802.3 MAC Sublayer Protocol

The MAC sublayer has two primary responsibilities:

Data encapsulation, including frame assembly before transmission, and frame parsing/error detection during and after reception **Media access control**, including initiation of frame transmission and recovery from transmission failure.

The frame structure is shown in Fig.22.14.

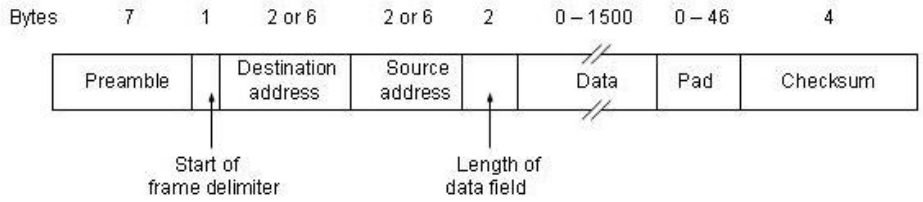

Figure 22.14: The 802.3 frame format

Preamble Each frame starts with Preamble of 7 bytes, each containing the bit pattern 10101010. This allows the receiver's clock to synchronize with the

sender's.

Start of frame delimiter It consists of 1 byte. It contains 10101011 and denotes the start of the frame itself.

Destination address It consists of 6 bytes. The high order bit of the destination address is a 0 for ordinary addresses and 1 for group addresses. Group addresses allow multiple stations to listen to a single address. When a frame is sent to a group address, all the stations in the group receive it. Sending to a group of stations is called multicast. The address consisting of all 1 bits is reserved for broadcast. A frame containing all 1s in the destination field is delivered to all stations on the network.

Source Address It consists of 6 bytes and identifies the sending station. The source address is always an individual address and the left-most bit of the source address is always 0.

Length This field tells how many bytes are present in the data field, from a minimum of 0 to a maximum of 1500. While a data field of 0 bytes is legal, it causes a problem. When a transceiver detects a collision, it truncates the current frame, which means that stray bits and pieces of frames appear on the cable all the time. To make it easier to distinguish valid frames from garbage, 802.3 states that valid frames must be at least 64 bytes long, from destination address to checksum. If the data portion of a frame is less than 46 bytes, the pad field is used to fill out the frame to the minimum size.

Data Is a sequence of n bytes of any value, where n is less that or equal to 1500. If the data portion of a frame is less than 46 bytes, the pad field is used to fill out the frame to the minimum size.

Pad It is used to fill out the frame to the minimum size.

Checksum It consists of 4 bytes. This field contains a 32-bit cyclic redundancy check (CRC) value, which is created by the sending MAC and is recalculated by the receiving MAC to check for damaged frames. The checksum is generated over the destination address, source address, length and data fields.

Another (and more important) reason for having a minimum length frame is to prevent a station from completing the transmission of a short frame before the first bit has even reached the far end of the cable, where it may collide with another frame. This problem is illustrated in the Fig.22.15.

At time 0, station A, at one end of the network, sends off a frame. Let us call the propagation time for this frame to reach the other end T. Just before the frame gets to the other end (i.e., at time T-E) the most distant station, B, starts transmitting. When B detects that it is receiving more power than it is putting out, it knows that a collision has occurred, so it aborts its transmission and generates a 48-bit noise burst to warn all other stations. At about time 2T, the sender sees the noise burst and aborts its transmission, too. It then waits for a random time before trying again.

If a station tries to transmit a very short frame, it is conceivable that a collision occurs, but the transmission completes before the noise burst gets back

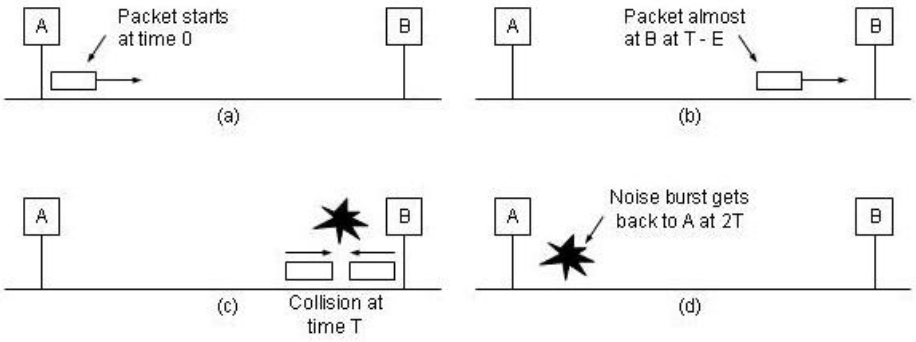

Figure 22.15: Collision detection can take as long as 2T

at 2T. The sender will then incorrectly conclude that the frame was successfully sent. To prevent this situation from occurring, all frames must take more than 2T to send. For a 10-Mbps LAN with a maximum length of 2500 meters and four repeaters (from the 802.3 specification), the minimum allowed frame must take 51.2 microseconds . This time corresponds to 64 bytes. Frames with fewer bytes are padded out to 64 bytes.

As the network speed goes up, the minimum frame length must go up or the maximum cable length must come down, proportionally. For a 2500-meter LAN operating at 1 Gbps, the minimum frame size would have to be 6400 bytes. Alternatively, the minimum frame size could be 640 bytes and the maximum distance between any two stations 250 meters.

Parameter	10 Mbps	100 Mbps	1000 Mbps
Minimum frame size	64bytes	64 bytes	520 bytes (with extension field added)
Maximum collision diameter, DTE to DTE	100 meters UTP	100 meters UTP 412 meters	100 meters UTP 316 meters fiber
Maximum collision diameter, with repeaters	2500 meters	205 meters	200 meters
Maximum number of, Repeaters in network path	5	2	1

Table 22.3: Limits for half-duplex operation

22.5 IP (Internet Protocol)

The Internet Protocol (IP) is a connectionless protocol of the network layer (layer 3 in the ISO/OSI network model). Connectionless means that a host can send a message without establishing a connection with the recipient first. That is, the host simply puts the message onto the network with the destination address and hopes that it arrives. IP is a datagram-oriented protocol, treating each packet independently. This means each packet must contain complete addressing information. Also, IP makes no attempt to determine if packets reach their destination or to take corrective action if they do not. Nor does IP check sum of the contents of a packet, only the IP header. IP protocol provides all of Internet's data transport services. Every other Internet protocol is ultimately either layered atop IP, or used to support IP from below. IP provides several services:

Addressing - IP headers contain 32-bit addresses which identify the sending and receiving hosts. These addresses are used by intermediate routers to select a path through the network for the packet.

Fragmentation - IP packets may be split, or fragmented, into smaller packets. This permits a large packet to travel across a network which can only handle smaller packets. IP fragments and reassembles packets transparently.

Packet timeouts - Each IP packet contains a Time To Live (TTL) field, which is decremented every time a router handles the packet. If TTL reaches zero, the packet is discarded, preventing packets from running in circles forever and flooding a network.

Type of Service - IP supports traffic prioritization by allowing packets to be labeled with an abstract type of service.

Options - IP provides several optional features, allowing a packet's sender to set requirements on the path it takes through the network (source routing), trace the route a packet takes (record route), and label packets with security features.
The header format is shown in Fig.22.16:

Version - Version field keeps track of which version of the protocol the datagram belongs to. By including the version in each datagram, it becomes possible to have the transition between versions take months, or even years, with some machines running the old version and others running the new one.
IHL - Since the header length is not constant, a field in the header, IHL, is provided to tell how long the header is, in 32-bit words. The minimum value is 5, which applies when no options are present. The maximum value of this 4-bit

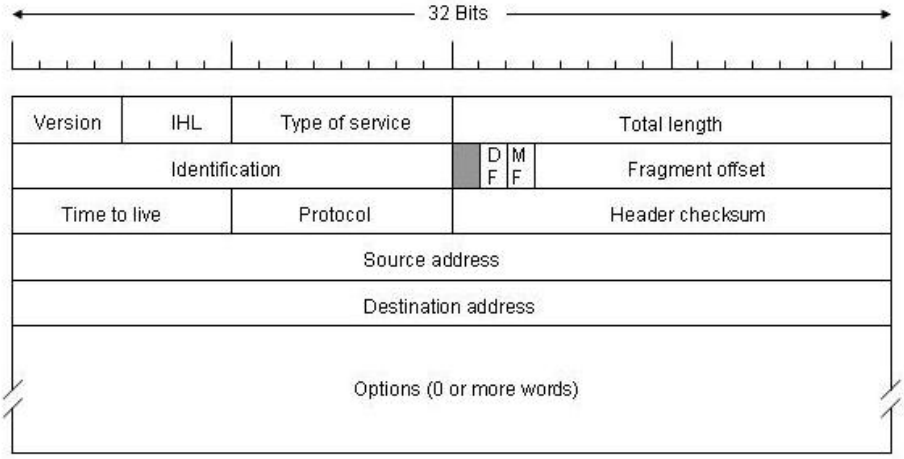

Figure 22.16: The IP (Internet Protocol) header

field is 15, which limits the header to 60 bytes, and thus the options field to 40 bytes. For some options, such as one that records the route a packet has taken, 40 bytes is far too small, making the option useless.

Type of service - The Type of service field allows the host to tell the subnet what kind of service it wants. Various combinations of reliability and speed are possible. For digitized voice, fast delivery beats accurate delivery. For file transfer, error-free transmission is more important than fast transmission.

Total length - This field includes everything in the datagram both header and data. The maximum length is 65,535 bytes. At present, this upper limit is tolerable, but with future gigabit networks larger datagrams may be needed.

Identification - The Identification field is needed to allow the destination host to determine which datagram a newly arrived fragment belongs to. All the fragments of a datagram contain the same Identification value.

Next comes an unused bit and then two 1-bit fields.

DF - DF stands for Don't Fragment. It is an order to the routers not to fragment the datagram because the destination is incapable of putting the pieces back together again. For example, when a computer boots, its ROM might ask for a memory image to be sent to it as a single datagram. By marking the datagram with the DF bit, the sender knows it will arrive in one piece, even if this means that the datagram must avoid a small-packet network on the best path and take a suboptimal route. All machines are required to accept fragments of 576 bytes or less.

MF - MF stands for More Fragments. All fragments except the last one have this bit set. It is needed to know when all fragments of a datagram have arrived.

Fragment offset - The Fragment offset tells where in the current datagram this

fragment belongs. All fragments except the last one in a datagram must be a multiple of 8 bytes, the elementary fragment unit. Since 13 bits are provided, there is a maximum of 8192 fragments per datagram, giving a maximum datagram length of 65,536 bytes, one more then the Total length field.

Time to live - The Time to live (TTL) field is a counter used to limit packet lifetimes. It is supposed to count time in seconds, allowing a maximum lifetime of 255 sec. It must be decremented on each hop and is supposed to be decremented multiple times when queued for a long time in a router. In practice, it just counts hops. When it hits zero, the packet is discarded and a warning packet is sent back to the source host. This feature prevents datagrams for wandering around forever, something that otherwise might happen if the routing tables ever become corrupted.

Protocol - When the network layer has assembled a complete datagram, it needs to know what to do with it. The Protocol field tells it which transport process to give it to. TCP is one possibility, but so are UDP and some others. The numbering of protocols is global across the entire Internet and is defined in RFC 1700.

Header checksum - The Header checksum verifies the header only. Such a checksum is useful for detecting errors generated by bad memory words inside a router. The algorithm is to add up all the 16-bit halfwords as they arrive, using one's complement arithmetic and then take the one's complement of the result. For purposes of this algorithm, the Header checksum is assumed to be zero upon arrival. This algorithm is more robust than using a normal add. Note that the Header checksum must be recomputed at each hop, because at least one field always changes (the Time to live field), but tricks can be used to speed up the computation.

Source address - The source host network number.

Destination address - The destination host network number.

Options - The Options field was designed to provide an escape to allow subsequent versions of the protocol to include information not present in the original design, to permit experimenters to try out new ideas, and to avoid allocating header bits to information that is rarely needed. The options are variable length. Each begins with a 1-byte code identifying the option. Some options are followed by a 1-byte option length field, and then one or more data bytes. The Options field is padded out to a multiple of four bytes. Currently five options are defined (but not all routers support all of them):

Security - specifies how secret the datagram is

Strict source routing - gives the complete path to be followed

Loose source routing - gives a list of routers not to be missed

Record route - makes each router append its IP address

Timestamp - makes each router append its address and timestamp

22.5.1 IP Addressing

Every computer that communicates over the Internet is assigned an IP address that uniquely identifies the device and distinguishes it from other computers on the Internet. An IP address consists of 32 bits, often shown as 4 octets of numbers from 0-255 represented in decimal form instead of binary form. For example, the IP address

168.212.226.204

in binary form is

10101000.11010100.11100010.11001100.

But it is easier for people to remember decimals than it is to remember binary numbers, so decimals are used to represent the IP addresses when describing them. However, the binary number is important because that will determine which class of network the IP address belongs to. An IP address consists of two parts, one identifying the network and one identifying the node, or host. The Class of the address determines which part belongs to the network address and which part belongs to the host address. All hosts on a given network share the same network prefix but must have a unique host number.

Class A Network – binary address start with 0, therefore the decimal number can be anywhere from 1 to 126. The first 8 bits (the first octet) identify the network and the remaining 24 bits indicate the host within the network. An example of a Class A IP address is 102.168.212.226, where "102" identifies the network and "168.212.226" identifies the host on that network. Class A allows for up to 126 networks with more than 16 million hosts each.

Class B Network – binary addresses start with 10, therefore the decimal number can be anywhere from 128 to 191. (The number 127 is reserved for loopback and is used for internal testing on the local machine.) The first 16 bits (the first two octets) identify the network and the remaining 16 bits indicate the host within the network. An example of a Class B IP address is 168.212.226.204 where "168.212" identifies the network and "226.204" identifies the host on that network. Class B allows for up to 16,382 networks with up to 65,534 hosts.

Class C Network – binary addresses start with 110, therefore the decimal number can be anywhere from 192 to 223. The first 24 bits (the first three octets) identify the network and the remaining 8 bits indicate the host within the network. An example of a Class C IP address is 200.168.212.226 where "200.168.212" identifies the network and "226" identifies the host on that network. Class B allows for up to 2 million networks with up to 254 hosts.

Class D Network – binary addresses start with 1110, therefore the decimal number can be anywhere from 224 to 239. Class D networks are used to support multicasting.

Class E Network – binary addresses start with 1111, therefore the decimal number can be anywhere from 240 to 255. Class E networks are used for experimentation. They have never been documented or utilized in a standard way.

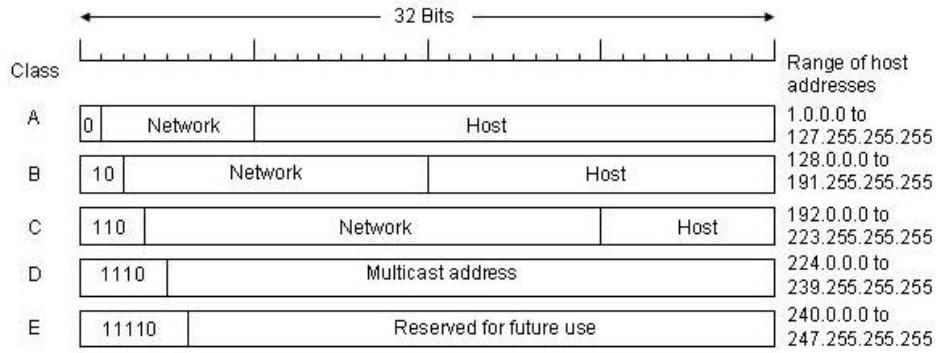

Figure 22.17: IP address formats

Identifying the host and the network part of IP address is not possible if only an IP address is given. Therefore, additional information is needed that would tell which part of the IP address belongs to the host part and which one to the network part. This additional information is called a subnet mask. It is composed of a 32 bit long binary number that starts with a series of 1s and ends with a series of 0s. Subnet mask and an IP address always come together. A subnet mask for Class C is 11111111 11111111 11111111 00000000 or in decimal form: 255.255.255.0. The number of 1s defines the network part of the corresponding IP address (in case of Class C address it is 24 1s). The number of 0s defines the host part of the address (in case of Class C address it is 8 0s). Performing a bitwise logical AND operation between the IP address and the subnet mask results in the Network Address or Number.
For example, using IP address 140.179.240.200 and the default Class B subnet mask, we get:

10001100.10110011.11110000.11001000 140.179.240.200 Class B IP Address
11111111.11111111.00000000.00000000 255.255.000.000 Class B Subnet Mask

---

10001100.10110011.00000000.00000000 140.179.000.000 Network Address

Default subnet masks:
Class A - 255.0.0.0 - 11111111.00000000.00000000.00000000
Class B - 255.255.0.0 - 11111111.11111111.00000000.00000000

Class C - 255.255.255.0 - 11111111.11111111.11111111.00000000

As noted above, IP address and subnet mask should always be given together. This can be done in two ways:

1. The subnet mask is given directly after the IP address, behind the slash (/) sign, for example: 192.168.17.1/255.255.255.0
2. Since the subnet masks always start with 1s and end with 0s (no 0 can be before a 1 in the subnet mask), the number of 1s defines the subnet mask. The same example as above can now be given as 192.168.17.1/24. This way of writing is called Classless InterDomain Routing (CIDR).

There are three IP network addresses reserved for private networks. The addresses are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 . They can be used by anyone setting up internal IP networks, such as a lab or home LAN behind a NAT or proxy server or a router. It is always safe to use these because routers on the Internet will never forward packets coming from these addresses. These addresses are defined in RFC 1918. The following table shows the ranges of these addresses:
Class A - 192.168.x.x
Class B - 172.16.x.x to 172.31.x.x
Class C - 10.x.x.x

The rest of IP addresses are unique in the internet world and the network administrators must ask for these IP addresses at Network Information Center (NIC), a central organization that takes care of delivering unique IP addresses to companies, organizations, etc. The table below shows the ranges of these, non-local addresses:
Class A - from 1.x.x.x to 9.x.x.x and from 11.x.x.x to 126.x.x.x
Class B - from 128.x.x.x to 172.15.x.x and from 172.32.x.x to 191.x.x.x
Class C - from 192.x.x.x to 192.167.x.x and from 192.169.x.x to 223.x.x.x

22.5.2 Subnetting

In 1985, RFC 950 defined a standard procedure to support the subnetting, or division, of a single Class A, B, or C network number into smaller pieces. Subnetting was introduced to overcome some of the problems that parts of the Internet were beginning to experience with the classful two-level addressing hierarchy:
Internet routing tables were beginning to grow.
Local administrators had to request another network number from the Internet before a new network could be installed at their site.

Both of these problems were attacked by adding another level of hierarchy to the IP addressing structure. Instead of the classful two-level hierarchy, subnetting supports a three-level hierarchy. Figure 6 illustrates the basic idea of

subnetting which is to divide the standard classful host-number field into two parts - the subnet-number and the host-number on that subnet.

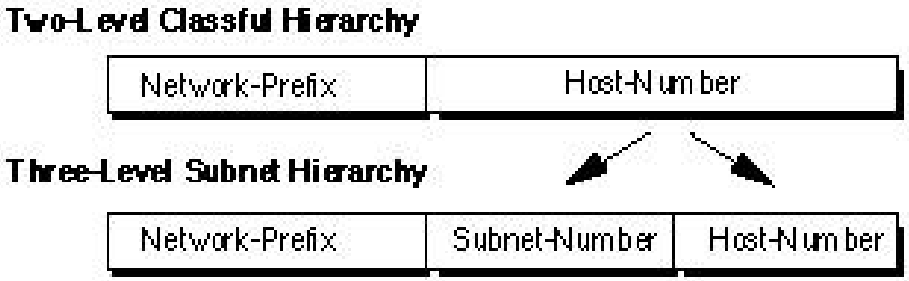

Figure 22.18: Subnet address hierarchy

Subnetting attacked the expanding routing table problem by ensuring that the subnet structure of a network is never visible outside of the organization's private network. The route from the Internet to any subnet of a given IP address is the same, no matter which subnet the destination host is on. This is because all subnets of a given network number use the same network-prefix but different subnet numbers. The routers within the private organization need to differentiate between the individual subnets, but as far as the Internet routers are concerned, all of the subnets in the organization are collected into a single routing table entry. This allows the local administrator to introduce arbitrary complexity into the private network without affecting the size of the Internet's routing tables.

Subnetting overcame the registered number issue by assigning each organization one (or at most a few) network number(s) from the IPv4 address space. The organization was then free to assign a distinct subnetwork number for each of its internal networks. This allows the organization to deploy additional subnets without needing to obtain a new network number from the Internet.

In Fig.22.19, a site with several logical networks uses subnet addressing to cover them with a single /16 (Class B) network address. The router accepts all traffic from the Internet addressed to network 130.5.0.0, and forwards traffic to the interior subnetworks based on the third octet of the classful address. The deployment of subnetting within the private network provides several benefits:

- The size of the global Internet routing table does not grow because the site administrator does not need to obtain additional address space and the routing advertisements for all of the subnets are combined into a single routing table entry.
- The local administrator has the flexibility to deploy additional subnets

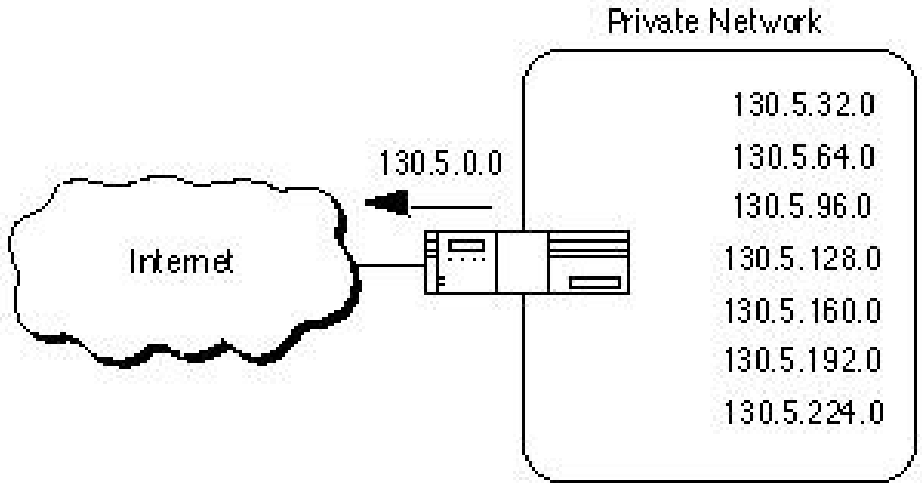

Figure 22.19: Subnetting reduces the routing requirements of the Internet

without obtaining a new network number from the Internet.

- Route flapping (i.e., the rapid changing of routes) within the private network does not affect the Internet routing table since Internet routers do not know about the reachability of the individual subnets - they just know about the reachability of the parent network number.

22.6 Internet Control Protocols

In addition to IP protocol, which is used for data transfer, the Internet has several control protocols used in the network layer, includeing ICMP and ARP.

22.6.1 The Internet Control Message Protocol (ICMP)

The operation of the Internet is monitored closely by the routers. When something unexpected occurs, the event is reported by the ICMP protocol, which is also used to test the Internet. ICMP protocol is documented in RFC 792. Some of ICMP's functions are to:

- Announce network errors, such as a host or entire portion of the network being unreachable, due to some type of failure. A TCP or UDP packet directed at a port number with no receiver attached is also reported via ICMP.

- Announce network congestion. When a router begins buffering too many packets, due to an inability to transmit them as fast as they are being received, it will generate ICMP Source Quench messages. Directed at the sender, these messages should cause the rate of packet transmission to be

slowed. Of course, generating too many Source Quench messages would cause even more network congestion, so they are used sparingly.

- Assist Troubleshooting. ICMP supports an Echo function, which just sends a packet on a round–trip between two hosts. Ping, a common network management tool, is based on this feature. Ping will transmit a series of packets, measuring average round–trip times and computing loss percentages.

- Announce Timeouts. If an IP packet's TTL field drops to zero, the router discarding the packet will often generate an ICMP packet announcing this fact. TraceRoute is a tool which maps network routes by sending packets with small TTL values and watching the ICMP timeout announcements.

About a dozen types of ICMP messages are defined and the most important ones are listed below:

Destination unreachable - Packet could not be delivered
Time exceeded - Time to live (TTL) field hit 0
Parameter problem - Invalid header field
Source quench - Choke packet
Redirect - Teach a router about geography
Echo request - Ask a machine if it is alive
Echo reply - Yes, I am alive
Timestamp request - Same as Echo request, but with timestamp
Timestamp reply - Same as Echo reply, but with timestamp

22.7 The Transmission Control Protocol (TCP)

The internet has two main protocols in the transport layer, a connection-oriented protocol (UDP) and a connectionless one (TCP). Because UDP is basically just an IP with a short header added.

TCP (Transmission Control Protocol) is a connection-oriented protocol that was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork (based on the connectionless IP protocol). An internetwork differs from a single network because different parts may have widely different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally was formally defined in RFC 793. As time went on, various errors and inconsistencies were detected, and the requirements were changed in some areas. These clarifications and some bug fixes are detailed in RFC 1122. Extensions are given in RFC 1323.

Each machine supporting TCP has a TCP transport entity, either a user process or part of the kernel that manages TCP streams and interfaces to the

IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64K bytes (in practice, usually about 1500 bytes), and sends each piece as a separate IP datagram. When IP datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just "TCP" to mean the TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in "The user gives TCP the data," the TCP transport entity is clearly intended.

The IP layer gives no guarantee that datagrams will be delivered properly, so it is up to TCP to time out and retransmit them as need be. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish the reliability that most users want and that IP does not provide.

22.7.1 The TCP Service Model

TCP service is obtained by having both the sender and receiver create and points, called sockets. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a port. To obtain TCP service, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine.

A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (socket1, socket2). No virtual circuit numbers or other identifiers are used.

Port numbers below 1024 are called well-known ports and are reserved for standard services. For example, any process wishing to establish a connection to a host to transfer a file using FTP can connect to the destination hosts port 21 to contact its FTP daemon. Similarly, to establish a remote login session using TELNET, port 23 is used. The list of well-known ports is given in RFC 1700.

All TCP connections are full-duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk, or some other way. There is no way for the receiver to detect the unit(s) in which the data were written.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As

with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte.

When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion. However, sometimes, the application really wants the data to be sent immediately. For example, suppose a user is logged into a remote machine. After a command line has been finished and the carriage return typed, it is essential that the line be shipped off to the remote machine immediately and not buffered until the next line comes in. To force data out, application can use the PUSH flag, which tells TCP not to delay the transmission.

Some early application used the PUSH flag as a kind of marker to delineate message boundaries. While this trick sometimes works, it sometimes fails since not all implementations of TCP pass the PUSH flag to the application on the receiving side. Furthermore, if additional PUSH-es come in before the first one has been transmitted (e.g., because the output line is busy). TCP is free to collect all the PUSHed data into a single IP datagram, with no separation between the various pieces.

One last feature of the TCP service that is worth mentioning here is urgent data. When an interactive user hits the DEL or CTRL-C key to break off a remote computation that has already begun, the sending application puts some control information in the data stream and gives it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms), so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked, so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out. This scheme basically provides a crude signaling mechanism and leaves everything else up to the application.

22.7.2 The TCP Protocol

Every byte on a TCP connection has its own 32-bit sequence number. For a host blasting away at full speed on a 10-Mbps LAN, theoretically the sequence numbers could wrap around in an hour, but in practice it takes much longer. The sequence numbers are used both for acknowledgements and for the window mechanism, which use separate 32-bit header fields.

The sending and receiving TCP entities exchange data in the form of segments. A *segment* consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the

65,535 byte IP payload. Second, each network has a *maximum transfer unit* or *MTU*, and each segment must fit in the MTU. In practice, the MTU is generally a few thousand bytes and thus defines the upper bound on segment size. If a segment passes through a sequence of networks without being fragmented and then hits one segment into two or more smaller segments.

A segment that is too large for a network that it must transit can be broken up into multiple segments by a router. Each new segment gets its own IP header, so fragmentation by routers increases the total overhead (because each additional segment adds 20 bytes of extra header information in the form of an IP header).

The basic protocol used by TCP entities is the sliding window protocol. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exists, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive. If the senders timer goes off before the acknowledgement is received, the sender transmits the segment again.

Although this protocol sounds simple, there are a number of sometimes subtle ins and outs that we will cover below. For example, since segments can be fragmented, it is possible that part of a transmitted segment arrives but the rest is lost and never arrives. Segments can also arrive out of order, so bytes 3072-4095 can arrive but cannot be acknowledged because bytes 2048-3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. If a retransmitted segment takes a different route than the original, and is fragmented differently, bits and pieces of both the original and the duplicate can arrive sporadically, requiring a careful administration to achieve a reliable byte stream. Finally, with so many networks making up the Internet, it is possible that a segment may occasionally hit a congested (or broken) network along its path.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implementations will be discussed below.

22.7.3 The TCP Segment Header

Fig.22.20 shows the layout of a TCP segment. Every segment begins with a fixed-format 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 - 20 - 20 = 65,495 data bytes may follow, where the first 20 refers to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

Let us dissect the TCP header field by field. The Source port and Destination port fields identify the local end points of the connection. Each host may

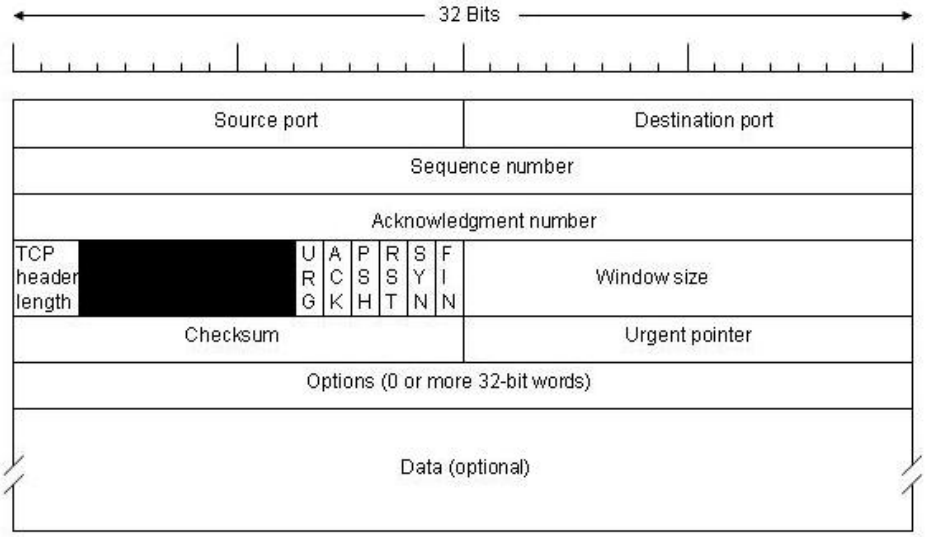

Figure 22.20: The TCP header

decide for itself how to allocate its own ports starting at 1024. The source and destination socket numbers together identify the connection.

The **Sequence number** and **Acknowledgement number** fields perform their usual functions. Note that the latter specifies the next byte expected, not the last byte correctly received. Both are 32 bits long because every byte of data is numbered in a TCP stream.

The TCP **header length** tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is too. Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 6-bit field that is not used. The fact that this field has survived intact for over a decade is testimony to how well thought out TCP is. Lesser protocols would have needed it to fix bugs in the original design.

Now come six 1-bit flags. URG is set to 1 if the **Urgent pointer** is in use. *The Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. As we mentioned above, this facility is a bare bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt.

The ACK bit is set to 1 to indicate that the *Acknowledgement number* is valid. If ACK is 0, the segment does not contain an acknowledgement so the *Acknowledgement number* field is ignored.

The PSH bit indicates PUSH-ed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency reasons).

The RST bit is used to reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection. In general, if you get a segment with the RST bit on, you have a problem on your hands.

The SYN bit is used to establish connections. The connection request has SYN=1 and ACK=0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has SYN=1 and ACK=1. In essence the SYN bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities.

The FIN bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, a process may continue to receive data indefinitely. Both SYN and FIN segments have sequence numbers and are thus guaranteed to be processed in the correct order.

Flow control in TCP is handled using a variable-size sliding window. The **Window size** field tells how many bytes may be sent starting at the byte acknowledged. A *Window size* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* - 1 have been received, but that the receiver is currently badly in need of a rest and would like no more data for the moment, thank you. Permission to send can be granted later by sending a segment with the same *Acknowledgement number* and a nonzero *Window size* field.

A **Checksum** is also provided for extreme reliability. It checksums the header, the data, and the conceptual pseudoheader shown in Fig.22.21. When performing this computation, the TCP *Checksum* field is set to zero, and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in 1s complement and then to take the 1s complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the *Checksum* field, the result should be 0.

The pseudoheader contains the 32-bit IP addresses of the source and destination machines, the protocol number for TCP (6), and the byte count for the TCP segment (including the header). Including the pseudoheader in the TCP checksum computation helps detect misdelivered packets, but doing so violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not the TCP layer.

The **Options** field was designed to provide a way to add extra facilities not covered by the regular header. The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept. Using large segments is more efficient than using small ones because the 20-

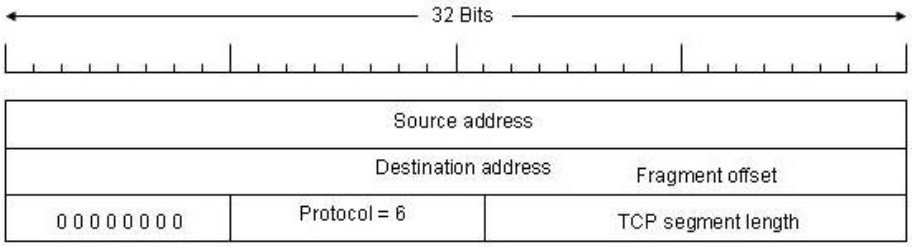

Figure 22.21: The pseudoheader included in the TCP checksum

byte header can then be amortized over more data, but small hosts may not be able to handle very large segments. During connection setup, each side can announce its maximum and see its partners. If a host does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segments of 536+20=556 bytes. The two directions need not be the same.

For lines with high bandwidth, high delay, or both, the 64 KB window is often a problem. On a T3 line (44,736 Mbps), it takes only 12 msec to output a full 64 KB window. If the round trip propagation delay is 50 ms (typical for a transcontinental fiber), the sender will be idle of the time waiting for acknowledgements. On a satellite connection, the situation is even worse. A larger window size would allow the sender to keep pumping data out, but using the 16-bit *Window size* field, there is no way to express such a size. In RFC 1323, a *Window scale* option was proposed, allowing the sender and receiver to negotiate a window scale factor. This number allows both sides to shift the *Window size* field up to 14 bits to the left. Most TCP implementations now support this option.

Another option proposed by RFC 1106 and now widely implemented is the use of the selective repeat instead of go back n protocol. If the receiver gets one bad segment and then a large number of good ones, the normal TCP protocol will eventually time out and retransmit all the unacknowledged segments, including all those that were received correctly. RFC 1106 introduced NAKs, to allow the receiver to ask for a specific segment (or segments). After it gets these, it can acknowledge all the buffered data, thus reducing the amount of data retransmitted.

22.7.4 TCP Connection Management

Connections in TCP are established using a three-way handshake. To establish a connection, one side, say the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives, either specifying a specific source or nobody in particular.

The other side, say the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection.

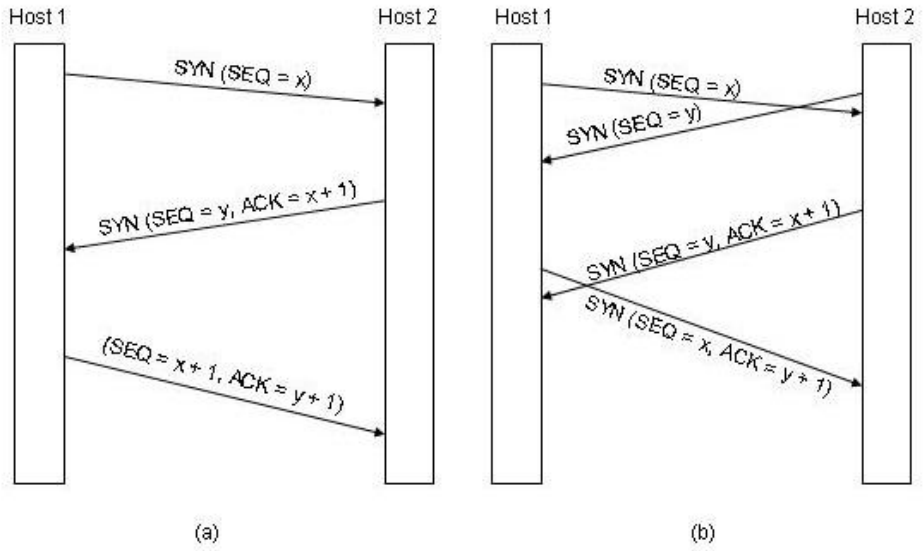

Figure 22.22: (a) TCP connection establishment in the normal case (b) Call collision

If some process is listening to the port, that process is given the incoming TCP segment. It can then either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig.22.22. Note that a SYN segment consumes 1 byte of sequence space so it can be acknowledged unambiguously.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in the Fig.22.22. The result of these events is that just one connection is established, not two because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

The initial sequence number on a connection is not 0 for the reasons we discussed earlier. A clock-based scheme is used, with a clock tick every 4 microsecond. For additional safety, when a host crashes, it may not reboot

for the maximum packet lifetime (120 sec) to make sure that no packets from previous connections are still roaming around the Internet somewhere.

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit. When the FIN is acknowledged, that direction is shut down for new data. Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection, one FIN and one ACK for each direction. However, it is possible for the first ACK and the second FIN to be contained in the same segment, reducing the total count to three.

Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send FIN segments at the same time. These are each acknowledged in the usual way, and the connection shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the so called two-army problem, timers are used. If a response to a FIN is not forthcoming within two maximum packet lifetimes, the sender of the FIN releases the connection. The other side will eventually notice that nobody seems to be listening to it any more, and time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Table 22.4.

22.7.5 TCP Transmission Policy

Window management in TCP is not directly tied to acknowledgements as it is in most data link protocols. For example, suppose the receiver has a 4096-byte buffer as shown in the Fig.22.23. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is 0. The sender must stop until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to make the receiver reannounce the next byte expected

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Table 22.4: The states used in the TCP connection management finite state machine

and window size. The TCP standard explicitly provides this option to prevent deadlock if a window announcement ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible. For example, in the last figure, when the first 2 KB of data came in, TCP, knowing that it had a 4 KB window available, would have been completely correct in just buffering the data until another 2 KB came in, to be able to transmit a segment with a 4 KB payload. This freedom can be exploited to improve performance.

Consider a TELNET connection to an interactive editor that reacts on every keystroke. In the worst case, when a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a 41-byte IP datagram. At the receiving side, TCP immediately sends a 40-byte acknowledgement (20 bytes of TCP header and 20 bytes of IP header). Later, when the editor has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also 41-byte packet. Finally, when the editor has processed the character, it echoes it as a 41-byte packet. In all, 162 bytes of bandwidth are used and four segments are sent for each character typed. When bandwidth is scarce, this method of doing business is not desirable.

One approach that many TCP implementations use to optimize this situation is to delay acknowledgements and window updates for 500 ms in the hope of

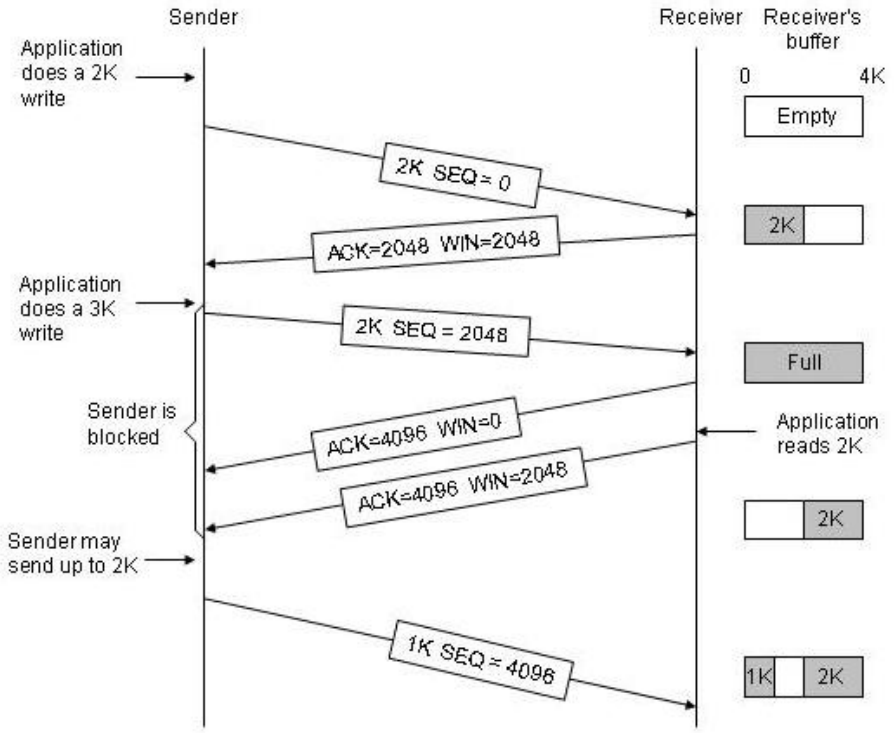

Figure 22.23: Window management in TCP

acquiring some data on which to hitch a free ride. Assuming the editor echoes within 500 ms, only one 41-byte packet now need be sent back to the remote user, cutting the packet count and bandwidth usage in half.

Although this rule reduces the load placed on the network by the receiver, the sender is still operating inefficiently by sending 41-byte packets containing 1 byte of data. A way to reduce this usage is known as *Nagles algorithm* (Nagle, 1984). What Nagle suggested is simple: when data come into the sender one byte at a time, just send the first byte and buffer all the rest until the outstanding byte is acknowledged. Then send all the buffered characters in one TCP segment and start buffering again until they are all acknowledged. If the user is typing quickly and the network is slow, a substantial number of characters may go in each segment, greatly reducing the bandwidth used. The algorithm additionally allows a new packet to be sent if enough data have trickled to fill half the window or a maximum segment.

Nagles algorithm is widely used by TCP implementations, but here are times when it is better to disable it. In particular, when an X-Windows application is being run over the Internet, mouse movements have to be sent to the remote computer. Gathering them up to send in bursts makes the mouse cursor move

erratically, which makes for unhappy users.

Another problem that can ruin TCP performance is the *silly window syndrome* (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data 1 byte at a time. To see the problem, look at the Fig.22.24. Initially, the TCP buffer on the receiving side is full and the sender knows this (i.e., has a window of size 0). Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment but sets the window to 0. This behavior can go on forever.

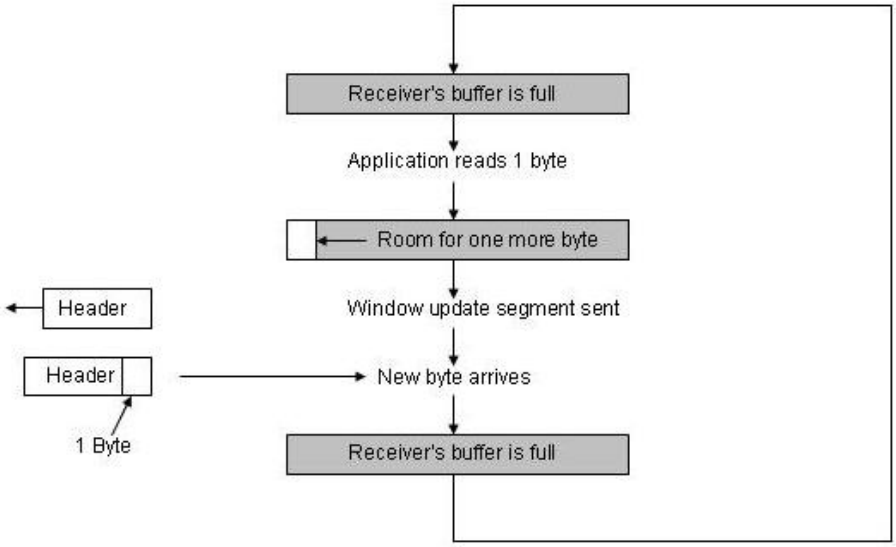

Figure 22.24: Silly window syndrome

Clarks solution is to prevent the receiver from sending a window update for 1 byte. Instead it is forced to wait until it has a decent amount of space available and advertise that instead. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established, or its buffer is half empty, whichever is smaller.

Furthermore, the sender can also help by not sending tiny segments. Instead, it should try to wait until it has accumulated enough space in the window to send a full segment or at least one containing half of the receivers buffer size (which it must estimate from the pattern of window updates it has received in the past).

Nagles algorithm and Clarks solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP byte at a time. Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

The receiving TCP can go further in improving performance than just doing window updates in large units. Like the sending TCP, it also has the ability to buffer data so it can block a READ request from the application until it has a large chunk of data to provide. Doing this reduces the number of calls to TCP, and hence the overhead. Of course, it also increases the response time, but for non-interactive applications like file transfer, efficiency may outweigh response time to individual requests.

Another receiver issue is what to do with out of order segments. They can be kept or discarded, at the receivers discretion. Of course, acknowledgements can be sent only when all the data up to the byte acknowledged have been received. If the receiver gets segments 0, 1, 2, 4, 5, 6, 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. If the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all byte up to the end of segment 7.

22.7.6 TCP Congestion Control

When the load offered to any network is more than it can handle, congestion builds up. The internet is no exception. In this section algorithms that have been developed over the past decade to deal with congestion, will be discussed. Although the network layer also tries to manage congestion, most of the heavy lifting is done by TCP because the real solution to congestion is to slow down the data rate.

In theory, congestion can be dealt with by employing a principle borrowed from physics: the law of conservation of packets. The idea is not to inject a new packet into the network until an old one leaves (i.e., is delivered). TCP attempts to achieve this goal by dynamically manipulating the window size.

The first step in managing congestion is detecting it. In the old days, detecting congestion was difficult. A timeout caused by a lost packet could have been caused by either (1) noise on a transmission line or (2) packet discard at a congested router. Telling the difference was difficult.

Nowadays, packet loss due to transmission errors is relatively rare because most long-haul trunks are fiber (although wireless networks are a different story). Consequently, most transmission timeouts on the Internet are due to congestion. All the Internet TCP algorithms assume that timeouts are caused by congestion and monitor timeouts for signs of trouble the way miners watch their canaries.

Before discussing how TCP reacts to congestion, let us first describe what

it does to try to prevent it from occurring in the first place. When a connection is established, a suitable window size has to be chosen. The receiver can specify a window based on its buffer size. If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.

The Internet solution is to realize that two potential problems exist - network capacity and receiver capacity - and to deal with each of them separately. To do so, each sender maintains two windows: the window the receiver has granted and a second window, the *congestion window*. Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows. Thus the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right. If the receiver says "Send 8K" but the sender knows that bursts of more than 4K clog the network up, it sends 4K. On the other hand, if the receiver says "Send 8K" and the sender knows that bursts of up to 32K get through effortlessly, it sends the full 8K requested.

When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends one maximum segment. If this segment is acknowledged before the timer goes off, it adds one segments worth of bytes to the congestion window to make it two maximum size segments and sends two segments. As each of these segments is acknowledged, the congestion window is increased by one maximum segment size. When the congestion window is n segments, if all n are acknowledged on time, the congestion window is increased by the byte count corresponding to n segments. In effect, each burst successfully acknowledged doubles the congestion window.

The congestion window keeps growing exponentially until either a timeout occurs o rthe receivers window is reached. The idea is that if burst of size, say, 1024, 2048, and 4096 bytes work fine, but a burst of 8192 bytes gives a timeout, the congestion window should be set to 4096 to avoid congestion. As long as the congestion window remains at 4096 no bursts longer than that will be sent, no matter how much window space the receiver grants. This algorithm is called *slow start*, but it is not slow at all (Jacobson, 1988). It is exponential. All TCP implementations are required to support it.

Now let us look at the Internet congestion control algorithm. It uses a third parameter, the *threshold*, initially 64K, in addition to the receiver and congestion windows. When a timeout occurs, the threshold is set to half of the current congestion window, and the congestion window is reset to one maximum segment. Slow start is then used to determine what the network can handle, except that exponential growth stops when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst) instead of one per segment. In effect, this algorithm is guessing that it is probably acceptable to cut the congestion window in half, and then it gradually works its way up from there.

Work on improving the congestion control mechanism is continuing. For example, Brakmo et al. (1994) have reported improving TCP throughput by 40 percent to 70 percent by managing the clock more accurately, predicting congestion before timeouts occur, and using this early warning system to improve the slow start algorithm.

22.7.7 TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the retransmission timer. When a segment is sent, a *retransmission timer* is started. If the segment is acknowledged before the time expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer started again). The question that arises is: How long should the timeout interval be? This problem is much more difficult in the Internet transport layer than in the generic data link protocols. In the later case, the expected delay is highly predictable (i.e. has low variance), so their timer can be set to go off just slightly after the acknowledgement is expected. Since acknowledgements are rarely delayed in the data link layer, the absence of an acknowledgement at the expected time generally means the frame or the acknowledgement has been lost.

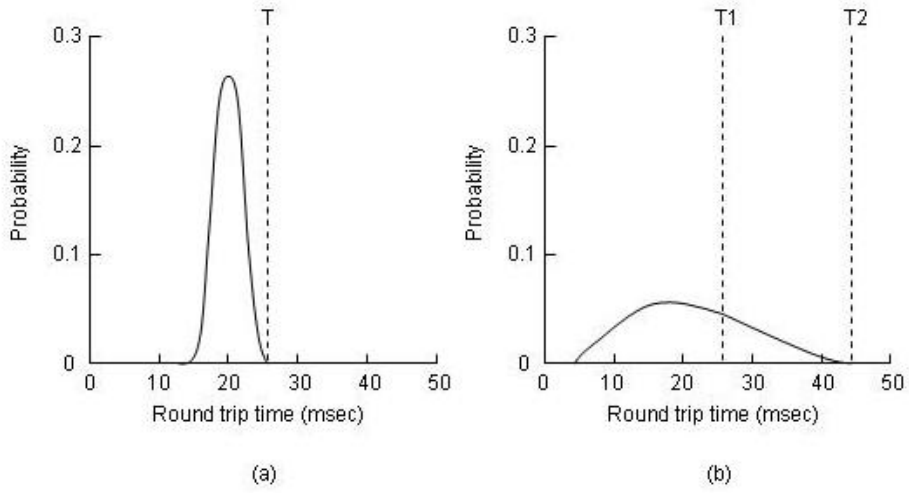

Figure 22.25: (a) Probability density of acknowledgement arrival times in the data link layer (b) Probability density of acknowledgement arrival times for TCP

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back

looks more like the Fig.22.25. Determining the round-trip time (RTT) to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say T1 in the Fig.22.25, unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long, (T2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

The solution is to use highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988).

One problem that occurs with the dynamic estimation of RTT is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the estimate of RTT. Phil Karn discovered this problem the hard way. He is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium (on a good day, half the packets get through). He made a simple proposal: do not update RTT on any segments that have been retransmitted. Instead, the timeout is doubled on each failure until the segments get through the first time. This fix is called *Karns algorithm*. Most TCP implementations use it.

The retransmission timer is not the only one TCP uses. A second timer is the *persistence timer*. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now both the sender and the receiver are waiting for each other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still zero, data can now be sent.

A third timer that some implementations use is the *keepalive timer*. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check if the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition.

The last timer used on each TCP connection is the one used in the TIMED WAIT state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

IMPORTANT:
A careful reader migth have already noticed that TCP protocol is not real-time capable because there is no mechanism implemented that would make it possi-

ble to control the time of packet-transmission. As soon as a fragment is lost for some reason and a retransmission timeout occurs, TCP automatically tries to resend this fragment. The application does not notice this directly nor it is able to influence the retransmission. Therefore, what you get is an unpredictable transmission behavior. TCP could be modified to be more deterministic but then this would not be TCP anymore.

22.8 The User Data Protocol (UDP)

The Internet protocol suite also supports a connectionless transport protocol, UDP (User Data Protocol). UDP provides a way for applications to send encapsulated raw IP datagrams and send them without having to establish a connection. Many client-server applications that have one request and one response use UDP rather than go to trouble of establishing and later releasing a connection. UDP is useful when TCP would be too complex, too slow, or just unnecessary. UDP is described in RFC 768.

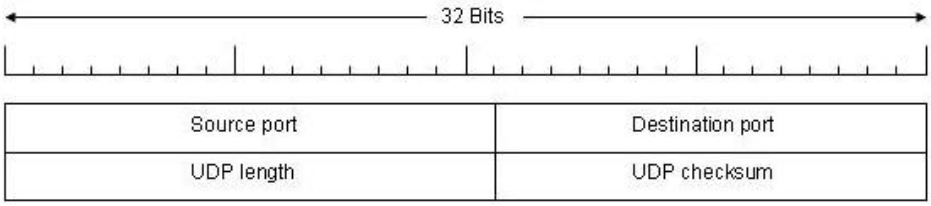

Figure 22.26: The UDP header

A UDP segment consists of an 8-byte header followed by the data. The header is shown in Fig.22.26. The two ports serve the same function as they do in TCP: to identify the end points within the source and destination machines. The *UDP length* field includes the 8-byte header and the data. The UDP also checksums its data, ensuring data integrity. A packet failing checksum is simply discarded, with no further action taken. This relatively large chapter covers protocol internals that we believe are necessary to be understood if different real-time networking implementations described later in the document are to be fairly evaluated. If the reader only wants to get an overview of the available real-time networking implementations or if she/he already posesses this knowledge, this chapter can be skipped. Otherwise it is strongly advised to read it through and get a good understanding of specifics of each of described protocols.

Chapter 23

Overview of Existing Extensions

23.1 rt_com

23.1.1 Overview and History

RT_Com is a serial port driver for RTAI and RTLinux. POSIX style interface layer is also available in RTLinux/GPL.
It was developed by Jens Michaelsen and Jochen Kuepper and POSIX layer has been added by Michael Barabanov.
Buffering is managed internally by the subsystem layer, in the rt_buf_struct structure that implements software FIFOs, used for buffering of the data that needs to be written to the port and data read from hardware that needs to be read by the user. The FIFO size is given by the define RT_COM_BUF_SIZ; which must be be a power of two, and must be configured at compile time.
RT_com API provides:
- serial port configuration
- serial port read/write Access
- call back function for serial interrupts
- internal managment functions for buffer managment

23.1.2 Guidelines

- **Official Homepage**
  http://rt-com.sourceforge.net
  http://sourceforge.net/projects/rt-com
- **Licensing**
  GPL for rt_com RTLinux/GPL
- **Availability of Source Code**
  Source code is available Integrated into RTLinux/GPL V3.2-preX

- **Supported RTOSi**
  - RTLinux/GPL up to 3.2 pre3
  - RTAI/RTHAL 2.24.X (obsolete for RTAI though - see spdrv)
  - Linux (non-rt)
- **Supported Kernel version**
  Fairly kernel version independant - what ever RTLinux/GPL supports will be supported by rt_com
- **Starting Date of the Project**
  - First release rt_com 1998
  - Sourceforge registration 14 Mar 2000
- **Latest Version**
  rt_com-0.5.5, 20 Jan 2002
- **Activity**
  Low (stalled)
- **Number of Active Maintainers**
  3
- **Supported HW Platforms**
  iX86
- **Supported Protocols**
  No protocol support, raw media
- **Supported I/O HW**
  16550 UART
- **Technical Support**
  Maling lists for rt_com:
  - RTAI mailing list at rtai.org
  - RTLinux at rtlinux.org

  are the main resource.
  Official mailing lists are at sourceforge.net:
  rt-com-devel@lists.sourceforge.net
  rt-com-user@lists.sourceforge.net
  The sorceforge mailing lists have very low activity though.
- **Applications**
  - Distributed embedded systems
  - Serial real-time devices
  - RTLinux/GPL - non-RT system communication

  NOTE: for current developments, usage of rt_com can not be recomended at this time as the project seems to be stalled and the roadmap for future development is unclear.

- **Reference Projects**

  - Robot Arm
    rt_com takes care of the real-time connection between the remote module (robot arm) and the controller module (joystick)
    http://www.opentech.at/ florianb
    Contact: Nicholas McGuire: der.herr@hofr.at, Georg Schiesser: georg_schiesser@gmx.at

- **Performance**
  No numbers published by developers (TODO: part2)

- **Documentation Quality**
  - API documentation: fair
  - Core technology documentation: none (source code only)
  - Examples: available, fairly complete

- **Contacts**
  - Jens Michaelsen: jens.michaelsen@web.de
  - Michael Barabanov (cross platform development): baraban@users.sourceforge.net
  - David Schleef: dschleef@users.sourceforge.net
  - Jochen Kuepper (everything that needs to be done): kuepper@users.sourceforge.net
  Note that this list of maintainers - all though the official list - is badly out of date...

23.2 spdrv

23.2.1 Overview and History

spdrv is a serial port driver for RTAI, simmetrically usable in kernel and user space applications that was developed by Paolo Mantegazza and Giuseppe Renoldi. It replaced rt_com for RTAI. spdrv offers backwards compatibility to older rt_com implementations under RTAI and it also offers serial port access to LXRT applications via an additional module rtai_spdrv_lxrt (which requires rtai_spdrv and rtai_lxrt to be loaded). spdrv API provides:

- serial port configuration
- serial port read/write Access
- call back function for serial interrupts
- internal managment functions for buffer managment

23.2.2 Guidelines

- **Official Homepage**
  No official homepage but sources can be retrieved from the RTAI CVS repository:
  http://cvs.rtai.org/index.cgi/stable/spdrv
- **Licensing**
  GPL for spdrv in RTAI
- **Availability of Source Code**
  - Source code is available
  - Integrated into current RTAI releases
- **Supported RTOS**
  - RTAI/ADEOS (to be considered experimental at time of writing) - RTAI/RTHAL 2.24.X
  - LXRT
- **Supported Kernel Version**
  2.4.X - basically anything supported by current RTAI/LXRT releases.
- **Starting Date of the Project**
  First release spdrv 2002 (CLEANUP: anything earlier ?version ??)
- **Latest Version**
  Seems like spdrv has no version numbers...
- **Activity**
  High, well maintained
- **Number of Active Maintainers**
  Officially 2, probably more
- **Supported HW Platforms**
  X86 (CLEANUP: others ?)
- **Supported Protocols**
  No protocol support, raw media
- **Supported I/O HW**
  16550 UART
- **Technical Support**
  RTAI mailing list at rtai.org
- **Applications**
  - Distributed embedded systems
  - Serial real-time devices
  - Serial communication between RT and non-RT systems

- **Reference Projects**

  - spdrv based interface between a PC and a robot,
    Katholieke Universiteit Leuven, Department of Mechanical Engineering (www.mech.kuleuven.ac.be)
    Contact: Herman Bruyninckx, Herman.Bruyninckx@mech.kuleuven.ac.be

  - Motion control system (Delta Tau PMAC controller)
    Project implemented by QA Technology Company, Inc. (www.qatech.com)
    spdrv is used to send ascii commands to a motion control system (Delta Tau PMAC controller) that is used for manufacturing. RTAI is used to run processes on the machines, and spdrv is used to communicate to the motion control in real time.
    Contact: Ken Emmons, Jr., kemmons@qatech.com

  - A project from the train industry, implemented by Envitech Automation (www.envitech.com).
    Project consists of two power units that were used as short-ciruit devices. The two units exchange status and information through a fiber-optic serial line driven by spdrv.
    Contact: Richard Brunelle, rbrunelle@envitech.com

  - Sychronization of RTAI tasks across a 422 network, project implemented by EMAC Inc. (www.emacinc.com).
    All the tasks are restarted when a specific serial character is received.
    Contact: Nathan Z. Gustavson, ngustavson@emacinc.com

- **Performance**
  No numbers published by developers (TODO: part2)

- **Documentation Quality**
  - API documentation: at least in the rtai releases the API is documented only in the source rtai_spdrv.c...
  - Core technology documentation: none (source code only)
  - Examples: available, fairly complete (ported from rt_com by Paolo Mantegazza)

- **Contacts**
  - Paolo Mantegazza mantegazza@aero.polimi.it (SPDRV core for RTAI)
  - Giuseppe Renoldi giuseppe@renoldi.org (extended LXRT)
  Note that this list of maintainers is probably not complete, but officially only these two are named.

23.3 RT-CAN

23.3.1 Overview and History

RTcan was developed by Seb James, a low level Linux programmer with Ph.D. in physics, who is incorporated in Hypercube Systems Ltd. He started developing rtcan in 2002, about a year ago due to the need for a real-time CAN network by one of his clients, Magnetic Systems Technology Ltd. RTcan is basically a set of functions that can be used for sending and receiving real-time CAN (Controller Area Network) messages from within RTAI threads. It was derived from ocan driver (version 0.13), which is a Linux CAN device driver for Intel's 82527 CAN controllers. ocan was developed by Alessandro Rubini (known for his book "Linux Device Drivers") and Rodolfo Giometti. Most of the changes have been done in memory management and in the interrupt routines and interrupt handler.You can get more information about ocan driver on http://www.linux.it/ rubini/software/#ocan.

Functions of RTcan are gathered in file libdrv.c. They can be called from within RTAI threads to send messages, and set up an interrupt handler to receive messages that need to be read from the receive buffer within another RTAI thread. RTcan is a new project therefore much of the development of rtcan is being done at the time of writing this document. RTcan has been developed on TQM8xxL target board but Seb James is currently porting RTcan to new platforms and is changing the structure of the software to make porting RTcan to new hardware platforms and CAN controllers easier.

Unfortunately, there is practically no documentation available about rtcan, but the author Seb James is very helpful at providing the necessary information.

23.3.2 Guidelines

- **Official Homepage**
  http://www.peak.uklinux.net/gnulin.php
  http://sourceforge.net/projects/rtcan
  Soon, the homepage for RTcan will be moved to
  www.hypercubesystems.co.uk/rtcan

- **Licensing**
  GPLv2

- **Availability of Source Code**
  Yes

- **Supported RTOS**
  RTAI Linux - it should work on any RTAI-kernel version, but so far it

hase been tested with RTAI 24.1.8 and RTAI 24.1.11 and Linux 2.4.4 on a TQM8xxL board

- **Supported Kernel Version**
  2.4.x, tested on kernel 2.4.4
- **Starting Date of the Project**
  Middle 2002
- **Latest Version**
  0.6 alpha, 13th July 2003
- **Activity**
  High
- **Number of active Maintainers**
  2
- **Supported HW Platforms**
  ix86 and PPC, it should be easy to port it to other platforms as well
- **Supported Protocols**
  CAN
- **Supported CAN Controllers**
  Intel 82527, Infineon 82c900, soon also Phillips sja1000 will be supported
- **Technical Support**
  No mailing list, only direct support by the author via E-mail:
  seb@hypercubesystems.co.uk
- **Applications**
  RTcan be used everywhere where a processor running Linux should send and receive CAN messages in real-time. Fields of applications are:
  - automotive industry
  - factory automation
  - machine control
  - building automation
  - medical applications
  - railway applications.
- **Reference Projects**
  According to Seb James, the author, he knows only of one project using RT-CAN:
  - vehicle management unit for a hybrid elcetric bus contains a RT-CAN module. Due to NDA (non disclosure agreement), Seb James could not provide more information about the project.
    Contact: Seb James, seb@hypercubesystems.co.uk

- **Performance**
  No available information (TODO: part2)

- **Documentation Quality**
  - API documentation: none
  - Core technology documentation: none, there is some documentation about ocan driver which was the basis for RTcan.
  - Examples: there is a very simple example included in the package that shows how to send a CAN message and how to put out a simple debug message.

- **Contacts**
  Seb James, author: seb@hypercubesystems.co.uk

23.4 RTnet

23.4.1 Overview and History

RTnet project was originally started in August 1999 by David Schleef who was at that time working for Lineo (now Metrowerks) and Lineo has publicly announced the availability of RTnet real-time networking solution in July 2000. At that time, RTnet was available for kernel 2.2 for both RTAI and RTlinux hard real-time extensions of Linux. In november 2001 Ulrich Marx, a student at the Institute for Systems Engineering at the University of Hannover, has reimplemented Schleef's concepts for his master's thesis and since then RTnet has been actively developed and maintained at this institute. Since the very beginning this project has been developed as an open-source project and was covered by the GPL license.

RTnet is basically a hard real-time protocol stack for RTAI (hard real-time Linux extension) that has been derived from the standard kernel TCP/IP stack. It offers a standard socket API to be used with RTAI kernel modules and LXRT processes. It is based on standard Ethernet hardware and supports several popular chipsets. IP, UDP, ICMP and ARP protocols are supported. Due to its nature TCP is not supported.

Network buffering has been reimplemented to match real time demands. According to the project leader Jan Kiszka, there is currently only a global rt-skbuff pool which delivers packet buffers for incomming and outgoing data. Plans exist to create a per-task based pool systems which would allow that even if one task does fail to return unused packets (overload etc.) other tasks would still be able to receive or send data. Therefore, currently a correct behaviour of all tasks is required.

Hard-real time capabilities of RTnet remain to be proven. So far, according to the maintainers of RTnet project, RTnet is behaving deterministically with fixed worst case latencies. But some hidden, not yet discovered bugs,could cause unexpected jitters in very special situations (hardware exceptions, interferences...). Only extensive testing and code studying could provide a definite answer.

RTnet also requires to have a full control over all transmissions in the networks to avoid collisions and congestions which means the network must be dedicated to the RTnet application.

"Standard" RTnet handles only IP (UDP) messages that fit into one IP frame (about 1400 byte UDP data) therefore an extension, called ipfragmentation, exists that enables RTnet to handle longer IP (UDP) packets. RTnet has been recently extended also with an additional protocol layer called RTmac that controls the media access and should prevent unpredictable collisions on the ethernet network.

rtnetproxy is an extension to RTnet that can be used to share a single network adapter for both - real-time and non real-time ethernet traffic. TCP/IP can be used via rtnet although not in real-time. Here is a picture from the RTnet docuementation:

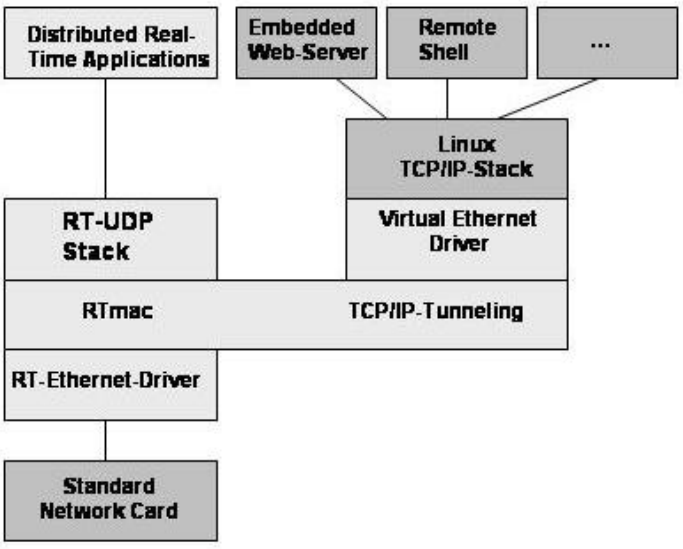

Figure 23.1: Internal structure of RTnet

23.4.2 Guidelines

- **Official Homepage**
  http://www.rts.uni-hannover.de/rtnet
  http://sourceforge.net/projects/rtnet

- **Licensing**
  GPLv2

- **Availability of Source Code**
  Yes

- **Supported RTOS**
  RTAI Linux - version 24.1.x

- **Supported Kernel Version**
  Latest kernel versions 2.4.x

- **Starting Date of the Project**
  2000 by David Schleef, in November 2001 it was reimplemented by Ulrich Marx

- **Latest Version**
  RTnet Version 0.2.10,27th June 2003

- **Activity**
  High

- **Number of Active Maintainers**
  - 10 listed on the Hannover homepage
  - 7 listed on the sourceforge homepage
  - 3-5 as reported by the project leader Jan Kiszka

- **Supported HW Platforms**
  Tested on x86 and PPC, should be compilable on other platforms as well

- **Supported Protocols**
  IP, UDP, ICMP, (statical) ARP

- **Supported NIC**
  - RealTek 8139
  - Intel EtherExpress Pro 100
  - 3Com EtherLink III
  - DEC 21x4x-based network adapters
  - MPC 8xx (SCC and FEC Ethernet)
  - MPC 8260 (FCC Ethernet)
  - rtnet_dev (rt-"loopback" device (rtnet_dev.c))

- **Technical Support**
  Active mailing list:
  Send mail to: rtnet-users@lists.sourceforge.net
  Subscribtion: http://lists.sourceforge.net/lists/listinfo/rtnet-users

- **Applications**
  - Cheap and fast field bus replacement for automation applications
  - Distributed real-time computing
  - Audio/video streaming

- **Reference Projects**
  Since the "new" RTnet (the one supported by the Hannover University) is a relatively new solution, there are not many projects that could be listed as reference projects. Most are still in the development phase and have not been published or advertised yet. Some more information about the ways RTnet is being used can be obtained through the following contacts:

  – Integration of RTnet into mobile robotic platforms,
    http://www.rts.uni-hannover.de/en/robots.htm,
    Contact: Jan Kiszka, jan.kiszka@web.de

  – A Remote Surveillance and Control System Prototype with RTLinux and RTNET,
    http://www.linuxdevices.com/articles/AT5207283655.html,
    Contact: Yan Shoumeng, ysminnpu@sina.com

  – Audio conferencing application using RTnet - a research project at the Appalachian State University (USA), Department of Computer Science,
    no URL of the project yet, only from the Department of Computer Science: http://www.cs.appstate.edu/,
    Contact: Shibu Vachery, vsy@cs.appstate.edu

  – Distributed automation application that requires guaranteed millisecond timing on the network.
    It's a commercial project, due to NDA between the customer and the contractor, no details could be provided.
    Contact: Michael D. Kralka, michael.kralka@kvs.com

- **Performance**
  As claimed by the project leader Jan Kiszka: Worst case latency (pentium platforms,
  IRQ→taskA→RTnet→taskB→digital output):
  Cycle time + 100...150 us
  Cycle time 5 ms → 8 Stations (Pentium 90)
  (50% load) → 20 Stations (Pentium MMX 266)
  Global time stamps: less than 50 us jitter

- **Documentation Quality**
  - API documentation: none
  - Core technology documentation: very little, only a general concept is described
  - Examples: there are examples in the RTnet package but they are not documented

- **Contacts**
  Jan Kiszka, project leader of RTnet: jan.kiszka@web.de

23.5 lwIP for RTLinux

23.5.1 Overview and History

lwIP is an open-source implementation of the TCP/IP protocol stack, developed with a focus on reducing memory usage and code size, making lwIP suitable for use in small clients with very limited resources such as embedded systems. lwIP can be used in systems with only tens of kilobytes of free RAM and ROM memory.

lwIP was originally written by Adam Dunkels at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS) for his master's thesis project but is now actively developed as an open-source project by a team of developers distributed world-wide. lwIP has been ported to several diferent hardware platforms and can be used with or without an underlaying operating system.

The layered protocol design of TCP/IP protocol stack has served as a guide for the design and implementation of lwIP. Each protocol is implemented as its own module, with a few functions acting asentry points into each protocol. Although the protocols are implemented separately, some layer violations are made in order to improve performance both in terms of processing speed and memory usage. Apart from modules implementing TCP/IP protocols some more modules are included in the lwIP package:

- operating system emulation layer, which provides a uniform interface to OS services such as timers, process synchronization and message passing mechanisms.
- buffer and memory management module
- network interface module
- module with functions for computing Internet checksum

lwIP provides two types of API:

- a specialized no-copy API for enhanced performance and
- a Berkeley Socket API

Later, in january 2003, lwIP was ported to RTLinux by Sergio Perez Alcaniz who named it RTL-lwIP.

RTL-lwIP includes IP, IPv6, ICMP, UDP and TCP protocols. It offers to real-time tasks a socket API to communicate with other real-time tasks or Linux processes over a network. RTL-lwIP inherits all lwIP's benefits and also adds new capabilities such as real-time capabilities and the characteristik of having an almost POSIX-compliant real-time operating system under it.
RTL-lwIP package also includes RTLinux drivers for the Ethernet cards 3Com905C-x and Realtek 8139 and a set of examples showing how to use RTL-lwIP.

23.5.2 Guidelines

- **Official Homepage**
  for lwIP:
  http://www.sics.se/ adam/lwip
  http://savannah.nongnu.org/projects/lwip
  for RTL-lwIP:
  http://canals.disca.upv.es/ serpeal/RTL-lwIP/htmlFiles/index.html
  http://bernia.disca.upv.es/rtportal/apps/rtl-lwip

- **Licensing**
  - BSD for lwIP
  - GPL for the RTLinux specific code of RTL-lwIP

- **Availability of Source Code**
  Yes

- **Supported RTOS**
  - lwIP can run with or without an underlaying OS. It has been ported to many RTOS so far, FreeBSD, Linux, MS-DOS, eCos are some of the best known examples
  - RTL-lwIP runs on RTLinux 3.1 and is already included in the RTLinux GPL 3.2-pre3 version

- **Supported Kernel Version**
  Latest supported kernel version: 2.4.18

- **Starting Date of the Project**
  - lwIP: 29th Jan 2001, when lwIP was innitially released
  - RTL-lwIP: 2nd Oct 2002, when Sergio Perez Alcaniz started porting lwIP to RTLinux

- **Latest Version**
  - lwIP: 0.6.4, 20th July 2003
  - RTL-lwIP: 0.3, May 2003, it is based on lwIP version 0.6.1
- **Activity**
  High
- **Number of Active Maintainers**
  - lwIP: 2
  - RTL-lwIP: 1
- **Supported HW Platforms**
  Porting lwIP to a new hardware platform should not be difficult, below are just some that are reported on the home page or in the mailing list of lwIP:
  - x86
  - 8051
  - Infineon C166 (ST10) platform with a SMsC LAN91C96 (or LAN91C94) Ethernet module
  - Mitsubishi M16
  - 68360

  If you want to use lwIP on RTLinux (RTL-lwIP), you can use it on all hardware platforms on which RTLinux is running (x86, PPC, StrongARM, ...).
- **Supported Protocols**
  IPv4, IPv6, ICMP, UDP, TCP
- **Supported NIC**
  - 3Com905C-X
  - Realtek8139
- **Technical Support**
  There is no special maling list for RTL-lwIP, the mailing list of lwIP is used instead.
  Send mail to: lwip-users@nongnu.org (it looks like the users mailing list is not alive, last mail in the list is from 13th July 2003)
  Subscribtion: http://mail.nongnu.org/mailman/listinfo/lwip-users
  Send mail to: lwip-devel@nongnu.org
  (developers mailing list is alive!)
  Subscribtion: http://mail.nongnu.org/mailman/listinfo/lwip-devel
- **Applications**
  - Distributed embedded systems
  - Real-time video and audio streaming

- **Reference Projects**
  There are a number of commercial and research projects using lwIP protocol stack, but due to a very recent port of lwIP to RTLinux only a few are using lwIP in combination with RTLinux (beside the author Sergio Perez Alcaniz we managed to find only one user of RTL-lwIP in the lwIP mailing list). Below are listed some commercial and research projects using pure lwIP:

  - Axon Digital Design BV in The Netherlands (http://www.axon.tv) is merging lwIP with their current IP stack for use in the Synapse modular broadcasting system, deployed at several broadcasters (such as BBC and CNN) and broadcast events (Formula 1 races).
    Contact: Leon Woestenberg, leon.woestenberg@axon.tv
  - UK based Tangent Devices Ltd (http://www.tangentdevices.co.uk) are incorporating lwIP in their film and video post-production equipment, which is planned to be used on the post-production of Lord of the Rings parts 2 and 3 among other films.
  - OpenFuel (http://www.openfuel.com) of South Africa are using lwIP in their Seth serial-to-Ethernet device
  - Arena Project (http://www.cdt.luth.se/projects/arena/) Pulse and breathing sensors running lwIP will be used by ice hockey players

- **Performance**
  No available information

- **Documentation Quality**
  There is plenty of good documentation about lwIP and also some on RTL-lwIP.
  - API documentation: complete both for lwIP and RTL-lwIP
  - Core technology documentaion: a good description of lwIP design issues in the Adam Dunkels' master thesis. Very little information about the changes that have been done by Sergio Perez Alcaniz in the RTL-lwIP.
  - Examples: there are some useful, but undocumented examples in the RTL-lwIP package.

- **Contacts**
  - lwIP:
  - Adam Dunkels, original author and the project leader of lwIP open-source project: adam@sics.se
  - Leon Woestenberg, administrator of the lwIP project site:
  leon.woestenberg@gmx.net - RTL-lwIP:
  - Sergio Perez Alcaniz, ported lwIP to RTLinux: serpeal@disca.upv.es

23.6 LNET/RTLinuxPro Ethernet

23.6.1 Overview and History

FSMLabs hard real-time network, LNET, intercepts the network connections passing all received data to a real-time handler. Packets destined for non real-time services managed by the general purpose OS (Linux) will be passed on when system resources are available (that is when no real-time task is ready to run). This concept of RTLinux hard real-time networking allows providing RT-networking over the same physical link that is used for the general purpose OS net-link. As Linux has all provisions for multi-homed systems providing a dedicated real-time link is simple. The decision if a dedicated link or a shared link is to be used is thus based on bandwidth and timing demands only.

Current ehternet hardware support is limited to the 3Com 3c905 fast ehternet chip, although the development effort for additional drivers is low (in the range of 2-4 weeks for a NIC supported by standard Linux). Packet latency is reported by fsmlabs to be below 85 micro seconds but details on the test-conditoins and network setup as well as system type and test-conditions are not available to judge how general these numbers are.

Support for IEEE 1394 is implemented in a generic fasion, hardware support extends to more or less all OHCI-1394 chipsets (VIA,TI,Lucent), the actual performance numbers though are naturally not independant of the underlaying hardware and can't be generalized. With TI chipsets round trip times under heavy load have shown jitter below 100 micro seconds. As details on the setup and the testconditions (system loads, network topology, effective utlized bandwith and operation modes (sync/async)) were not available at time of writing these number must be concidered preliminary.

LNET utilizes the standard Linux facilities to initilize the hardware (PCI functions of Linux for NIC and IEEE-1394 PCI initialisation). The LNET implementation is a layered model providing

- buffering, application specific - responsibility of the application programmer.
- signaling, RX and TX handlers are supported for notification/callback functions.
- header/envelope managment via macros (up to the programmer to know at what offset what is in the header....)

in a common layer. The hardware driver is an independantly loaded module and encapsulates the hardware specifics of the ethernet/firewire hardware only, whereby the ethernet driver is based on the Linux 3C905 driver, conversely the

firewire driver is a reimplemented IEEE 1394 layer, more or less from scratch. Buffering is user managed, error codes are provided - no error handling on the LNet subsystem layer LNET API is POSIX Socket style API but is also non POSIX complient.

23.6.2 Guidelines

- **Official Homepage**
  http://www.fsmlabs.com

- **Licensing**
  Commercial license: license@fsmlabs.com
  Non-commercial academic licenses: restricted

- **Availability of Source Code**
  Binary only available (per seat license)
  Source code available (per seat license)

- **Supported RTOS**
  RT-Core/Linux (RTLinux/Pro)
  RT-Core/BSD

- **Supported Kernel Version**
  2.4.7 and 2.4.16
  Kernels must be pre-patched with RT-Core patches

- **Starting Date of the Project**
  June 2001 by FSMLabs Inc.

- **Latest Version**
  LNet 1.0 (Dev-Kit 1.3)

- **Activity**
  High, but only one maintainer

- **Number of active maintainers**
  1: weaver@fsmlabs.com

- **Supported HW Platforms**
  Tested on x86 and PPC, documentation and numbers pertain to x86 though (CLEANUP: check ARM and MIPS)

- **Supported Protocols**
  IPv4, RAW IP, UDP (via sockets), ICMP

- **Supported NIC**
  - 3Com EtherLink III (3c90X)
  - eepro/1000 (no support for eepro/100 !)
  - National Semiconductor DP83815

- **Technical Support**
  - Commercial support offerings by FSMLabs Inc. USA.
  - Currently no non-commercial support offerings nore support offering other than by FSMLabs are known.

- **Applications**
  - Distributed control applications
  - Data aquisition systems

- **Reference Projects**
  Pratt and Whitney CLEANUP: details
  CLEANUP: others ?? details

- **Performance**
  - Worst case latency:
  - Cycle time:
  - Global time stamps:
  (CLEANUP: FSMLabs agreed to provide 'official' data by the end of August 2003) (TODO: part2)

- **Documentation quality**
  - API documentation: man-pages, complete, well readable
  - Core technology documentation: incomplete, "marketing" quality only
  - Examples: "self-explaining code", no concept documentation available
  This assesment pertains to the documentation provided with the RTLinux/Pro Dev-Kit and only takes documents into account that are included in the release or referenced by the material included in the release. External documentation may be available but is not known off at the time of writing.

- **Contacts**
  support@fsmlabs.com
  FSMLabs Inc., 914 Paisano Drive Socorro NM 87801, USA
  Maintainer: weaver@fsmlabs.com

23.7 LNET/RTLinuxPro 1394 a/b

23.7.1 Overview and History

Firewire is one of the best suitedst low-cost technologies available in the area of distributed realtime systems, it is to be expected that firewire support will also imerge in other variants of realtime enhanced Linux, and especially be

enhanced in the standard Linux kernel. Conceptually IEEE 1394 (a/b) are much beter suited for realtime networking problems than IP based protocols. Firewire does not have the irritating problem of fragmentation in the transport layer, and it has provisions for both asynchronous and synchronous (timesliced) operations, which clearly fits the problems of distributed realtime systems better than CSMA/CD and CSMA/CA based systems (including CAN).

At the time of writing the only available firewire drivers for realtime enhanced Linux are FSMLabs LNet drivers.

LNET 1394:

LNet 1394 drivers support isocronous and asynchronous transfer. Multichannel mode is supported for one socket/device. Since real-time operations are disturbed during a bus reset operation, it is the programmers responsibility to react to such events. The underlaying LNET-subsystem does provide bus managment functions but can't guarantee any timings on bus-reset events (nodeIDs can change and the reset operation that can be triggered by any node will disrupt any transfere for time in the range of up to tens of seconds.

LNET can control multiple OHCI1394 devices. The driver has the following features:

- support for asynchronous requests and responses
- support for isochronous stream packets
- support for asynchronous (a.k.a. loose) stream packets
- supports up to 32 isochronous transmit contexts
- supports up to 32 isochronous receive contexts
- Cycle Master capability
- Isochronous Resource Manager (IRM) capability
- limited Bus Manager capability w/ topology map
- read/write access to local PHY registers
- read/write access to local IRM registers
- read access to the local topology map
- ability to tune any iso rx context to listen to a specific channel or enter multichannel mode
- ability to tune any iso tx context to transmit on a specific channel
- IRQ sharing between devices using this driver ONLY

- bus reset notification
- support for up to 63 nodes on a bus
- support for up to 16 ports per node

Buffer allocation
As the headers of asynchronous packets are not constant and the maximum MTU is dependant on the wire speed there is no simple static buffer structure. The buffer allocation and management is left to the application programmer. Buffer management is provided in the API via fcntl functions on the associated devices/sockets.
API
The hardware initialization utilizes the Linux PCI functionality as found in th standard kernel, the LNET firewire API is a socket based API providing:

- socket creation/binding
- read/write access
- filter settings (on node basis)
- restricted bus-managment functionality
- control functions via fcntl
- packet header access via get/set macros

23.7.2 Guidelines

- **Official Homepage**
  http://www.fsmlabs.com
- **Licensing**
  - Comercial per-node license
  - There also is a license on the developer-seat for developers and LNET requires use of RTLinux/Pro.
  - Academic licenses are available (although no academic projects are known...)
- **Availability of Source Code**
  Yes, but only as a commercial product
- **Supported RTOS**
  RTLinux/Pro
- **Supported Kernel Version**
  2.4.7 and 2.4.16

- **Starting Date of the Project**
  CLEANUP:

- **Latest Version**
  LNET 2.0, Aug 2003

- **Activity**
  High, actively maintained

- **Number of active Maintainers:**
  1

- **Supported HW Platform**
  iX86

- **Supported Protocols**
  IEEE 1394 a and b

- **Supported I/O HW**
  Any OHCI 1394 device seems to work. Tested are:
  - TI chipset based OHCI 1394 cards
  - VIA OHCI 1394 based cards
  - Lucent technology OHCI 1394 cards
  As long as the hardware is OHCI 1394 complient it should be expected to work, a detailed list of actuall cards could not be obtained. No details on 1394b support could be obtained at this point, 1394b support has though not yet been released (due within the next few weeks - info weaver@fsmlabs.com).
  NOTE: Tested means 'pluged in and did not fail' - no long term testing done with the full set of devices.

- **Technical Support**
  - Commercial support offerings by FSMLabs Inc. USA.
  - Currently no non-commercial support offerings nore support offering other than by FSMLa

- **Applications**
  - Distributed data-aquisition systems (Simens SiCOM project)
  - Distributed controll applications
  - Supervisor applications (wire sniffing)

- **Reference Projects**
  -

- **Performance**
  (CLEANUP: data should be available within the next few days - FSMLabs agreed to provide 'extensive' data) (TODO: part2)

- **Documentation**
  As IEEE 1394 is a very complex bus the technological basis is not documented in the framework of the LNET implementation but references to external documents are provided.
  - API documentation: man-pages, complete, well readable
  - Core technology documentation: incomplete, "marketing" quality only
  - Examples: "self-explaining code", no concept documentation available

- **Contacts**
  support@fsmlabs.com
  FSMLabs Inc., 914 Paisano Drive Socorro NM 87801, USA

23.8 REDD/Real Time Ethernet Device Driver

23.8.1 Overview and History

23.8.2 Guideline

- **Official Homepage**
  REDDs homepage is at sourceforge, as of version 0.4 REED has been merged into the main RTLinux/GPL development. It is actively maintained at:
  http://www.rtlinux-gpl.org
  (http://www.rtlinux-gpl.org/cgi-bin/viewcvs.cgi/rtlinux-3.2-rc1/drivers/redd/)

- **Licensing**
  - GPL V2

- **Availability of Source Code**
  Yes

- **Supported RTOS**
  RTLinux 3.2-rc1

- **Supported Kernel version**
  2.4.X (up to 2.4.30)

- **Starting Date of the Project**
  early 2004

- **Latest Version**
  0.4 released Dec, 2004

- **Activity**
  High

- **Number of active Maintainers**
  - Sergio Perez Alcaniz, author: serpeal@upvnet.upv.es
  - Nicholas McGuire, maintainer of the RTLinux GPL track: der.herr@hofr.at
- **Supported HW Platforms**
  Tested on x86, it should work on other platforms as well since it does not use any platform specific code.
- **Supported Protocols**
  RAW IP only
- **Technical Support**
  Mailing list at redd.sourceforge.net and rtlinuxgpl@upv.es
  aswell as web-resources at www.rtlinux-gpl.org.
  Send mail to: rtlinuxgpl@upv.es
  Subscription: https://listas.upv.es/mailman/listinfo/rtlinuxgpl
- **Applications**
  Used in the academic field.
- **Reference Projects**
  - TODO
- **Performance**
  No available information (TODO: part2)
- **Documentation Quality**
  - Instalation and usage documentation incomplete
  - API documentation: POSIX conform API
  - Core technology documentaion: fully documented
  - Examples: basic examples included in distribution.
- **Contacts**
  - Sergio Perez Alcaniz ¡serpeal@upvnet.upv.es¿ author, - Nicholas McGuire, maintainer of the RTLinux GPL track: der.herr@hofr.at

23.9 RTsock

23.9.1 Overview and History

Author of RTsock solution is Robert Kavaler, who started his work on RTsock in 1998 as part of a product development for a communications product. Since that product was never put on the market, Robert Kavaler decided to initialy release RTsock on 20th Jan 2000. Later, RTsock project was merged into the GPL track of RTlinux, maintained by Nicholas McGuire.

RTsock allows you to comunicate via sockets from rt-threads to non-rt processes locally or on remote systems - the comunication is non real-time but it allows comunication via networks without requiring to copy data to user space via fifo/mbuff and then moving it over the network.

RTsock is not a device driver for network cards, it is an RTLinux interface to the Linux sockets. The main advantage is that all of the standard layer 2 and layer 3 protocols already implemented in the Linux kernel are available to the real-time task meaning all Linux routing protocols, ifconfigs, ARP, RARP, QoS, netfiltering and other packet level processes are applied to the real-time socket. Packets flow through the Linux kernel using the standard Linux drivers, up/down the standard layer 2 and layer 3 protocols, and then packets are diverted into an RTLinux task.

The disadvantage of this approach is that the Linux kernel is executed as the lowest priority task (it is executed only when no real-time task need to be executed) the conesquence of wich is that RTsock would cause unpredictable delays when packets are received and sent through the Linux kernel. Another disadvantage of RTsock is that only UDP sockets are supported.

23.9.2 Guideline

- **Official Homepage**
  RTsock alone does not have any official home page, but was merged into the main RTLinux/GPL development. It is actively maintained at:
  http://www.rtlinux-gpl.org
  (http://www.rtlinux-gpl.org/cgi-bin/viewcvs.cgi/rtlinux-3.2-pre3/network/rtsock/)
- **Licensing**
  - BSD
  - GPL V2 when used with the GPL version of RTLinux
- **Availability of Source Code**
  Yes
- **Supported RTOS**
  RTLinux 3.2-pre3
- **Supported Kernel version**
  2.4.19/20/21
- **Starting Date of the Project**
  January 2000, when RTsock was initialy released
- **Latest Version**
  1.1 released April 29, 2003

- **Activity**
  High

- **Number of active Maintainers**
  - Robert Kavaler, author: kavaler@berkeley.innomedia.com
  - Nicholas McGuire, maintainer of the RTLinux GPL track: der.herr@hofr.at

- **Supported HW Platforms**
  Tested on x86 and PPC, it should work on other platforms as well since it does not modify the linux drivers

- **Supported Protocols**
  IP, UDP

- **Technical Support**
  No mailing list dedicated only to RTsock, but you can us RTLinux mailing list at www.rtlinux.org.
  Send mail to: rtl@rtlinux.org
  Subscription: http://www2.fsmlabs.com/mailman/listinfo.cgi/rtl

- **Applications**
  As claimed by the author: "The main application for the rtsock interface is in situations that require real-time generation or consumption of standard UDP packets in an otherwise asynchronous network. One example is a time-tick that must be generated at a fixed rate to a large group of machines. Another is for RTP sessions in a VoIP application, when the generator/consumer is a DSP or Video card running with a constant clock, but the network side is a standard ethernet. In this case, a "jitter buffer" must be implemented in the real-time task."

- **Reference Projects**

  – Research of transfering sensor samples via UDP packets from the sensor to the computing node in real-time. Project is still in a very early stage so no URL is available.
  contact: Narayan, narayan@deeproot.co.in

  – Research project at Stirling Dynamics (www.stirling-dynamics.com), conducted by Stephen Brown. Its aim is to replace the existing 16bit RTOS, used on control sticks for flight simulators, with RTlinux. RTsock is used for communication of the control stick module with the Windows front end application via UDP protocol.
  Contact:Stephen Brown, lcbr17306@blueyonder.co.uk

- **Performance**
  No available information (TODO: part2)

- **Documentation Quality**
  All the documentation consists of one 200 lines long document.
  - API documentation: there is a section about how to use RTsock
  - Core technology documentaion: practicaly none
  - Examples: there is one, well documented example included in the rtsock source code.

- **Contacts**
  - Robert Kavaler, author: kavaler@berkeley.innomedia.com
  - Nicholas McGuire, maintainer of the RTLinux GPL track: der.herr@hofr.at

23.10 TimeSys Linux/Net

23.10.1 Overview and History

TimeSys enhanced their Linux distribution for embedded systems and made it preemptive. They also improved the control over interrupt handling. The distribution is called TimeSys Linux GPL, and the sources of this distribution are freely available under the terms of a GPL license. TimeSys claims that their preemptive kernel provides better average and maximum latency times than the new (version 2.5.4 and later) kernel. They claim to achieve average latency of 50 microsec and maximum latency of 1000 microsec with their GPL distribution.

However, since these enhacements alone are not sufficient for a true real-time performance, TimeSys provides extensions to their modified GPL kernel that provide better controls over resource allocation, scheduling, and usage accounting. These extensions are available as loadable kernel modules and their sources are NOT available under the terms of a GPL license. These modules are called: *TimeSysLinux/Real-Time*, *TimeSysLinux/CPU* and *TimeSys Linux/NET*. According to TimeSys, these extensions reduce the latency times even more, down to 10 microsec for average latency times and 51 microsec for maximum latencies. The primary idea behind these modules are so called reservations.
As opposed to priorities which are enabling high priority threads and interrupt handlers to be executed before lower priority ones, a reservation represents a share of a single computing resource. Such a resource can be CPU time (as in TimeSys Linux/CPU), network bandwidth (as in TimeSys Linux/NET), physical memory pages, or a disk bandwidth. TimeSys provide two types of reservations: CPU Reservations (module TimeSys Linux/CPU) and Network Reservations (module TimeSys Linux/NET).

As claimed in the Timesys documentation:
"A CPU reservation is a guarantee that a certain amount of CPU time will be available to a thread (or a set of threads) at a defined periodic rate, no matter what else is happening in the system, and independent of the priorities of other

threads. For example, using CPU reservations, the thread can request a reservation for six milliseconds of CPU time out of every 300 milliseconds of elapsed time. Such a request could be hard to fullfil by using only priority mechanism. There are two kinds of CPU reservations: - hard reservations: the thread will never get more than the amount of CPU time reserved in each period. - soft reservations: the thread will compete for the CPU at a low priority when its reservation is exhausted in a given period."

Similar to CPU reservations, "a network resevation guarantees the ability to send and/or receive certain number of bytes on a periodic basis and it also ensures that sufficient buffer space is made available for receipt of incoming packets for reserved. Network reservations consist of two separate capabilities: they can control bytes received at a socket, bytes sent to a socket, or both. For example, a sensor management thread might reserve 1.2 KB to be received every 26 miliseconds, and 363 bytes to be sent every 420 miliseconds. In such an example, the incoming and outgoing packets would be handled using the standard Linux IP stack, but the scheduling of the IP stack components would be controlled by the reservation mechanism to ensure that the bandwidth is avialable in both directions. Network reservation mechanism does not directly control the network itself, but rather controls access to the network hardware by controlling the sequence of IP processing within the Linux kernel to ensure that the reservation is honored. This means that incoming packets destined for a reserved thread will be separately queued for transmission to the network adapter by the device driver in priority order."

23.10.2 Guidelines

- **Official Homepage**
  http://www.timesys.com

- **Licensing**
  - GPL for the standard kernel, modified by TimeSys and named TimeSys Linux GPL
  - TimeSys End-user License for modules *TimeSysLinux/Real-Time*, *TimeSysLinux/CPU* and *TimeSys Linux/NET* - No run time royalties need to be paid.

- **Availability of Source Code**
  Source code for TimeSys Linux GPL is freely available, loadable kernel modules *TimeSysLinux/Real-Time*, *TimeSysLinux/CPU* and *TimeSys Linux/NET* are only available as binary modules. Source code for these modules can be provided under certain conditions that need to be agreed with TimeSys company.

- **Supported RTOS**

Real-time extensions (loadable kernel modules *TimeSysLinux/Real-Time*, *TimeSysLinux/CPU* and *TimeSys Linux/NET*) only work with the TimeSys Linux GPL distribution.

- **Supported Kernel Version**
  Different for different hardware plaforms, currently the x86 release is for 2.4.18 version.

- **Starting Date of the Project**
  Development of Linux/NET was started in 1998.

- **Latest Version**
  4.0, released in March 2003

- **Activity**
  High

- **Number of active Maintainers**
  TimeSys is willing to reveal this information only in the final stage of the purchase.

- **Supported HW Platforms**
  - ARM7, ARM9
  - StrongARM (1110, IXP1200)
  - IA-32 (x86, Pentium I, II, III)
  - MIPS (MIPS32 Au1500, CorelV 4Kc, CorelV 5Kc, VR4122, VR5432, VR5500)
  - SuperH (SH-3 7709A, SH-4 7750S)
  - PowerPC (750, 7400, 7410, 8240, 8245, 823, 8260, 850, 855, 860)
  - UltraSparc (UltraSPARC 11e)
  - XScale (80200, 80310, 80321, PXA250)

- **Supported Protocols**
  IP, UDP, TCP - reservations work on a socket level so basically all protocols that are based on sockets, are supported.

- **Technical Support**
  Technical support is only available as a paid service. TimeSys Technical Support consists of reasonable e-mail and telephone support during normal U. S. business hours, U. S. Eastern Standard Time (excluding U. S. holidays and weekends), bug-fixes, updates, and Technical Support Web Site access.

- **Applications**
  - Telecommunications - in switch control plane processing, running failure notifications under a reservation can ensure that failure cascading doesn't stop call management functions during critical situations.

- Car navigation - running the satellite functions under a reservation can ensure that they are not impacted by display or other, less critical, activities.
- Air traffic control - using a reservation to control confliction detection can ensure that sensor returns, weather information, and other resource updates (e.g., runway closings, navigation aid maintenance) will not result in unsafe conditions even when the system is under stress.
- Process control - reservations can separate critical functions such as sensor/actuator management from less time-critical functions, ensuring safe operation without requiring physical resource separation.

- **Reference Projects**
  TimeSys is willing to reveal this information to qualified purchasers of TimeSys Linux/NET technology in the final stages of the purchase. Though, they did provide us a link to the announcement of NASA using TimeSys Linux OS and their Real-Time Java technology on Mars Exploration Rover:
  http://www.timesys.com/index.cfm?bdy=home_bdy_news.cfm&show_article=125

- **Performance**
  Latency times (10 us average, 51 us maximum), provided by TimeSys, are only available as values, without detailed description of conditions under which these values have been measured. (TODO: part2)

- **Documentation Quality**
  - API documentation: complete Core technology documentation: plenty, but all on a "marketing" level.
  - Core technology documentation: concept of reservations is described, but no detailed information on the actual implementation in the TimeSys Real-Time, CPU and NET modules
  - Examples: did not find any examples in the TimeSys Linux GPL package that can be donwloaded freely. No information about examples for Rea-Time, CPU and NET modules but it is assumed that such examples exist and are availabe in the purchased package.

- **Contacts**
  Thomas Vincent, sales representative:
  vincent.thomas@timesys.com Inquiries:
  http://www.timesys.com/index.cfm?bdy=inquire.cfm
  call 1-412-232-3250 or 1-888-432-8463

Chapter 24

Conclusion

In this chapter we will try to summarize some of the basic conclusions about what real-time networking medium is best stuited for what situation. These conclusions can never be valid for all situations but they should hold valid for most and provide some guidlines for the design phase of new projects that require network connectivity. It must be noted though that the results here are based on analysis of the underlaying technology and published performance data. Due to the very limited availability of performance data and especially the low quality of this performance data (except for RTnet in all cases the environment of the tests were not given at all or only incomplete - in most cases not even the hardware used and the network topology were given), a firm statement on these issues is not posible without appropriate test/benchmarks and are to be concidered preliminary.

The networking subsystems are categorized into:

- hard real-time networking
- soft real-time (QOS) networking
- non real-time connectivity to the real-time subsystem

These three categorizatioins are focused on latency - although real-time networking can also be focused on guaranteed bandwidth allocation this is not concidered here simply because none of the implementations available target this issue and provide any usable data on this issues - TODO PART2: benchmark/analyze bandwidth properties of available implementations.

24.1 Hard Real-Time Networking

Ordered by the preference we see for actual application is the list of available hard real-time networking extensions. Networking is generally viewed as a protocol based communication between at least two nodes therefore strictly speaking

serial lines hardly fall into the category of networking. But due to limitations in the real-time capabilities of existing (especially IPv4) protocols that limit these to an extent where they are hardly are offering much more than raw data conectivityit, seems justified to treat serial lines as a valid real-time networking facility.

The order of concidering real-time networking soultions when trying to fulfill hard real-time requirements, is seen bellow:

- serial lines for point to point
- firewire for larger systems and systems that need the bandwidth (available for dumb-nodes aswell -¿ frauenhofer inst. firewire stack for microcontrolers...with no error handling...)
- CAN as alternative for distributed real-time systems, especially for "dumb nodes"
- ethernet for real-time - non real-time communication but not for real-time systems
- other (parport (CLEANUP: check HSD))

The order given is based on

- availability of technology (especially in the embedded world)
- cost of technology
- reliability and community experience with the technology
- programming simplicity
- performance

It should be noted though that naturally before selecting a specific technology one needs to know timing and bandwidth demands of the application. Especially with respect to bandwith the listed technologies vary greatly.

24.1.1 Preference for serial lines:

Sounds like old technology - but it does the job for many actual products and thus should be concidered first. Advantages of serial lines are:

- simple
- well tested and robust
- inexpensive

- easy to debug and validate with external equipment
- available on almost any target architecture
- available on most SBC's and NN-PC systems

24.1.2 Preference for firewire:

- one driver for all (OHCI standard)
- high bandwidth
- no fragmentation
- deterministic bandwidth assignment
- large systems posible - very flexible with concern to topology
- inexpensive
- expandable

24.1.3 Preference for RT-CAN:

- well tested
- design concepts for prioritized networks available
- CAN interfaces on many deeply embedded devices available
- robust technology

24.1.4 Usage of ethernet as hard real-time networking infrastructure:

Even though it may seem like a resonable approach to take an inexpensive, well tested media like 802.3 to build hard real-time networks there are a number of issues that need to be considered and in our view theses issues make 802.3 a not too attractive solution to the problem of real-time networking.

Drawbacks of real-time ethernet:

- bus arbitration
- protocol capabilities
- data managment issues
- header issues

Header issues:

- no time stamp
- no priority
- header complexity, designed for routing, not used
- routing capabilities are not real-time safe

Other open issues:

- sharing media (possibilities provided will be used and thus must be safe)
- EMV reliability of ethernet on the factory floor

24.2 Soft Real-Rime (QoS) Networking

Linux networking code has developed a large number of QoS strategies that are merged into the main stream kernel by now. A real comparison of all existing implementations is not available at this point (TODO: phase 2). QoS strategies, available in Linux, are naturally inherently soft real-time approaches. Implementations include:

- CBQ packet scheduler (known to be problematic in its current implementation in 2.4.X kernels and has a fundamentaly poor delay characteristics)
- HTB packet scheduler [**?**]
- CSZ packet scheduler [**?**]:
  The Clark-Shenker-Zhang (CSZ) packet scheduling algorithm for some of your network devices. At the moment, this is the only algorithm that can guarantee service for real-time applications - each guaranteed service is provided with a dedicated flow with pre-allocated resources. This strategy is somewhat inflexible and is not very efficient in terms of overall bandwidth utilisation, but it can guarantee good worst case performance.
- simplest PRIO pseudo-scheduler:
  Uses an n-band priority queue packet "scheduler" as a disciplin that can be assigned within a CBQ scheduling algorthing (leaf discipline) see note on CBQ above.
- SFQ scheduling algorithm (another CBQ leaf disciplin)
- TEQL queue (CBQ leaf discipline) that allows chanel bonding in itself. It is not a soft real-time QOS approach but a way of increasing bandwidth

- TBQ queue for CBQ. It tries to provide a comparable aproach as HTB within CBQ (which seems useless due to the inherently bad latency and jitter of CBQ...)

- scheduling algorithm based on Differentiated Services architecture (proposed in RFC 2475) [39]

A further interesting aspect of QoS in mainstream Linux is that when sharing networks between real-time and non real-time, the available packet classification API [40] allows to limit the bandwidth of non real-time trafic very selectively thus (posibly) improving real-time reliability (TODO: phase 2 verify/validate/benchmark shared links with restricted non real-time trafic).

24.3 Non Real-Time Connectivity to Real-Time Threads

In the category of non real-time networking between Linux and real-time enhanced Linux one must distinguish two categories:

- utilization of standard Linux networking capabilities

- dedicated solution

24.3.1 Standard Linux Networking

In this study we are interested in the dedicated solutions only, and the meainstream Linux networking capabilities are well documented so repeating these here seems to make little sense. It should be noted though that for quite a few applications the Linux networking infrastructure is more than suited and as a general rule:

- don't use special solutions if mainstream solutions will do

- average performance will always be beter with mainstream Linux than with dedicated solutions

- security issues are best solved in mainstream Linux networking implementations

Normal non real-time networking can thus be seen as an advanced IPC between real-time and non real-time nodes (comparable to FIFOs from user-space to real-time threads).

24.3.2 Dedicated non Real-Time Networking

The implementations listed here are for RTLinux/GPL

- RT-sock
- RTL-lwIP

Although for RTLinux/GPL,they are not very specific to this hard real-time extension. Especially RT-sock should be trivial to port to any of the other impementations (at the time of writing, this has not happened yet though).

As lwIP requires specific drivers for the network card (that is it can't use standard Linux drivers) and the advantage of using lwIP with respect to system memory footprint is not very impressive, we see little insentive to base a project on lwIP at this point. Works to allow hard real-time networking connections via lwIP have been proposed but this seems not to be very realistic due to the buffering being done by the subsystem and not the application layer. It is our belive that buffering must be explicidly managed by the application to allow garanteed hard real-time behavior of the system.

RT-sock is the preferable implementation to conect real-time threads via sockets to remote systems at this point (non real-time networking). RT-sock is able to utilize the available Linux networking code including Linux networking drivers which makes it very flexible and easy to maintain. The layer is basically the socket API expanded to the kernel side by providing wrappers to the socket related system call functions. Basically all that can be done with RT-sock can be done by passing data to user space (i.e. via real-time fifos) and then using the regular socket API. This is though fairly expensive and thus for low resource systems (slow processors). Using RT-sock has a clear advantage of reducing the system load. A further effect of RT-sock usage is the simplification of the networking related system components as the intermediate step to user space is not required.

For non real-time socket connectivity of real-time threads to remote systems, RT-sock is to be concidered the preferable choice of method.

Chapter 25

Resources

Books:
- Andrew S. Tanenbaum: Computer Networks, third edition , 1996, Prentice-Hall
- Dietmar Dietrich, Wolfgang Kastner, Thilo Sauter: EIB Gebaeudebussystem, 2000, Huethig Verlag Heidelberg
- Herman Kopetz: Real-Time Systems - Design Principles for Distributed Embedded Applications, 1997, Kluwer Academic Publishers
- Andrew S. Tanenbaum: Modern Operating System (2nd Edition), 2001, Prentice-Hall

Documents:
- An assessment of real-time robot control over IP networks,
G. H. Alt,R. S. Guerra, W. F. Lages, Federal University of Rio Grande do Sul,Electrical Engineering Department, Porto Alegre Brazil, Proceeding of the 4th Real Time Linux Wokshop

URLs:

spdrv:
http://cvs.rtai.org/index.cgi/stable/spdrv

RT_com:
http://rt-com.sourceforge.net
http://sourceforge.net/projects/rt-com
http://www.mrao.cam.ac.uk/ dfb/doc/rtlinux/MAN/rt_com.3.html

RT-CAN:
http://www.peak.uklinux.net/gnulin.php
http://sourceforge.net/projects/rtcan
http://www.linux.it/ rubini/software/index.html#ocan

http://www.linux.it/ rubini/software/ocan/ocan.html
http://www.can-bus.com/can/en
http://212.114.78.132/can
http://www.hypercubesystems.co.uk
http://www.can.bosch.com

RTnet:
http://www.rts.uni-hannover.de/rtnet
http://sourceforge.net/projects/rtnet
http://www.linuxdevices.com/articles/AT5207283655.html
http://www.linuxdevices.com/news/NS4023517008.html
http://www.emlix.com/en/opensource/rtnet
ftp://ftp.linuxincontrol.org/pub/events/licws-2003/slides/rtnet.pdf

RTsock:
http://www.rtlinux-gpl.org/cgi-bin/viewcvs.cgi/rtlinux-3.2-pre3/network/rtsock
ftp://ftp.opentech.at/pub/rtlinux/contrib/kavaler/README
http://canals.disca.upv.es/lxr/source/network/rtsock-1.1

RTL-lwIP:
http://www.sics.se/ adam/lwip
http://savannah.nongnu.org/projects/lwip
http://canals.disca.upv.es/ serpeal/RTL-lwIP/htmlFiles/index.html
http://bernia.disca.upv.es/rtportal/apps/rtl-lwip
http://www.hurray.isep.ipp.pt/rtlia2003/full_papers/5_rtlia.pdf

LNET/RTLinuxPro Ethernet:
http://www.fsmlabs.com
http://www.fsmlabs.com/products/lnet/lnet.html

LNET/RTLinuxPro 1394 a/b:
http://www.fsmlabs.com
http://www.fsmlabs.com/products/lnet/lnet.html
http://www.linuxdevices.com/news/NS8806718594.html

TimeSys Linux/NET:
http://www.timesys.com
http://www.timesys.com/index.cfm?hdr=tools_header.cfm&bdy=tools_bdy_time.cfm
http://www.timesys.com/index.cfm?hdr=sdk_header.cfm&bdy=sdk_bdy_platforms.cfm
http://www.realtime-info.be/vpr/layout/display/pr.asp?PRID=3014
http://www.eetimes.com/story/OEG20020621S0075
http://www.timesys.com/index.cfm?bdy=home_bdy_news.cfm&show_article=125

RTcan:

http://www.peak.uklinux.net/gnulin.php
http://sourceforge.net/projects/rtcan
http://www.linux.it/ rubini/software/index.html#ocan
http://www.linux.it/ rubini/software/ocan/ocan.html
http://www.hypercubesystems.co.uk

RS232:
http://www.ctips.com/rs232.html
http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html
http://www.sangoma.com/signal.htm

IEEE 1394:
http://www.1394ta.org
http://www.computer.org/multimedia/articles/firewire.htm
http://www.embedded.com/1999/9906/9906feat2.htm

CAN bus:
http://www.can-bus.com/can/en
http://212.114.78.132/can
http://www.can.bosch.com

Ethernet:
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ethernet.htm
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/bridging.htm
http://www.yale.edu/pclt/COMM/ETHER.HTM

IP protocol:
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm
http://www.ralphb.net/IPSubnet
http://www.3com.com/other/pdfs/infra/corpinfo/en_US/501302.pdf

ICMP protocol:
http://www.freesoft.org/CIE/Topics

TCP protocol:
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm
http://penguin.dcs.bbk.ac.uk/academic/networks/transport-layer/tcp/index.php
http://www.freesoft.org/CIE/Course/Section4/
http://www.ssfnet.org/Exchange/tcp/tcpTutorialNotes.html

UDP protocol:
http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm

Part IV

Overview of embedded Linux resources

25.1 Introduction

As Embedded GNU/Linux systems are establishing themselves as solid alternative to many proprietary OS and RTOS, not only because they allow the developer to use comodity components in embedded system (off the shelf PC's), and open a large software resource on the internet, but simply because GNU/Linux has become a reliable and mature embedded OS with available RTOS extension, thus covering the entire range of embedded devices. Demands on embedded Linux developers and providers are increasing as capabilities are available that give rise to security issues as well as system interoperability as more and more embedded systems are using the available Internet infrastructure. In this part an attemt is made to scetch the top requirements/problems for embedded GNU/Linux systems and give an overview of resource available for these demands. The main chalanges will be to fit contradicting demands into embedded systems - this demands are:

- Simple end-user interface vs. in depth diagnostic and administrative interface
- High level of security vs. open and simple access to the system via network and local interfaces.
- Resource contstraints vs. high system complexity and low response time, real time capabilities being one of the comon demands in embedded systems.

The information presented in this article is the destilat of the embedded Linux/RTLinux activities which the authors were involved in over the past years. As the focus was on very small 32bit systems targeting real time applications and distributed systems, there naturally is a slight slant towards that end here - nevertheless this is an attempt at giving an overview for the practician and a basis for making design descisions. Embedded GNU/Linux can offer solutions satifying the contradictiong demands noted above and at the same time expanding the potential application field of embedded OS/RTOS if the advanced capabilities of GNU/Linux are taken into account from the very beginning of the design phase.

Embedded Linux distributions have been around for quite a while. Single floppy distributions, mainly targeting the X86 architecture, like the Linux router project (lrp) and floppy firewall are well known by now. This first step into embedded Linux distributions was accompanied by a fair amount of 'home-brew' embedded Linux variants for custom devices, expanding the architecture range into powerpc MIPS, SH, ARM and others. Embedded Linux is more and more becoming a usable and easy to handle Linux segment. But what is the position of embedded Linux? Where does it fit into the other embedded OS in the 32bit market? In this article a few thoughts on the 'why embedded Linux?'

will be sketched out, as I belive, positioning embedded Linux quite high up on the list of first-choice embedded OS and RTOS.

A side theme of this section is to scan the potential of Dev-Kits evolving for the multitude of embedded Linux platoforms, how usable they are and what potential problems/limitations they might incure. This is hardly done with a simple check list so by introducing the spectrum of resources available for embedded linux systems the concludion can focus on the issue of developement enviroments and development kits.

25.2 The main chalenges in Highend Embedded OS

What are the main challenges for system designers and programmers in the embedded world ? The list given is definitely not complete and reflects lots of personal impressions - it is thus only one view under many - heated debates on what is required may be fought on mailing lists. Consider the following as 'one picture', hopefully offering constructive thoughts on the subject, even if not all may be applicable to some systems.

25.2.1 User Interface

A major point of criticism of embedded Linux systems is their lack of a simple user interface - generally embedded systems have an archaic touch to their user interface. But a tendency that is evolving is to split the user interface into three distinct sections

- The actual user-interface allowing to control the systems dedicated application providing a system overview along side that gives you a general 'system up and running' or 'call the technician' information.
- A in depth interface that allows you to configure/update and diagnose system operations at an expert level from the application specifics all the way down to OS's internals.
- The log facility that allows long term tendency analyzing as well as backtracking of events in case of a fatal error (i.e. when the embedded system was not able to respond appropriately)

This split is not always done cleanly and is not always visible to the user, it will often run on one interface, but this split is anticipated by most interfaces of embedded devices - representing the actual operational demands. Simple to use for common operations - clear and instructive to the maintenance personnel in case of errors and long term data that can be processed independently of the current status of the specific device. Embedded Linux can provide all three in a very high quality if designed to these goals from the very beginning on. Many

embedded Linux distributions offer a web-server giving OS-independent remote access to status information - at the same time maintenance via secure shell can allow insight into the system down to directly poking around in the kernel at runtime without disturbing the systems operation and simple inter-operability with other networked OS's allows off-site logging and tendency analysis.

Operational Interface

HMI's as machin-tool designers like to call it or GUI's as OS developers will prefer are some sort of generally graphical based interface that should allow close to untrained personnel to inter-operate with specialized hard and software. A problem that arises here is that embedded systems are limited in available resources and fully developed X-Windows systems are very greedy with respect to RAM and CPU usage (if anybody tried out XFree 4.0 on a 486 without FPU...at 33MHz let me know how long the window-manager takes to "launch"). So does this mean forget embedded Linux if you need a graphical interface ? Nop - there are quite a lot of projects around , nano-X, tyni-X, and projects that give you direct access to the graphics display like libsvga or frame-buffer support in recent kernels. Getting an acceptable graphics interface running on an embedded Linux platform is still a challenge even though IBM has shown that one can run XClock on top of XFree86 in a system with no more than 8MB footprint, generally a 32MB storage device and 16MB RAM will be the bottom line (there are some PDA distributions though that are below that). The Operator Interface will be a simple scale down variant of a "standard" Linux desk-top in many cases and this simplifies development greatly as the graphics libraries available for Linux cover a very wide range - with a new widget set emerging every few weeks.

Aside from this console interface a networked interface can provide the operator with all required input/output functionality with a minimum on local resourses,shifting resource demands from the embedded system to a OS-independent interface that will run on ANY remote system - with any ranging from a desk-top PC to a mobile phone.

Administrative Interface

Embedded Products have traditionally required skilled personnel to handle error situations or performance/setup issues. This basically is due to a non-standard operating-system model behind all these devices. The goal was to have a intuitive interface at the expert level (and many hours of training...) which limited the potential scope of intervention and at the same time raised maintenance costs of such devices. Embedded Linux takes a different approach - you have a very large and seemingly complete operator interface - a more or less complete UNIX clone - and this allows operators to debug, analyze and intervene with great precession at the lowest level of the GNU/Linux OS. The advantage is

clear - you don't need to learn each product - it's a GNU/Linux system just like a multiprocessor-cluster, a web-server or a desk-top system - one interface for the entire range of possible applications. This allows operators and technicians to focus on the specifics of each platform without great training efforts on a per-device basis. Even though the initial investment in training can be relatively high - all attempts to manage complex problems using simple interfaces are severely limited - POSIX II gives a complex and powerful interface to the operator that allows adequate response to a complex and powerful embedded operationg system.

Status and Error reporting

Checking the status of the fax-machine or an elevator is not a high-end administrative task and should not require any knowledge of details at all. To this end Linux offers the ability to communicate with users directly via the console (simply printk'ing errors on a text-console) or a web-interface as well as offering a OS-independent active response via voice, email, SMS or turning on a siren if one connects it to some general output pin of the system. So the resources required for clean status and error reporting are available in Linux and embedded Linux but care must be taken as to what information can be displayed in response to errors as this naturally touches security issues. Error messages need to be clear and status information needs to be informative - "An application error occurred - [OK]" is not very helpful - on the other hand it is not always desirable if error messages include the exact version of the OS/Kernel/application and the TCP port on which it is listening... as this could reveal information that allows attacking such a system.

Early fault detection

Embedded systems may easily die silently - error-nous behavior is detected only when the services it should provide are requested and the system does not respond - but how do you figure out what happened. Many common failure scenarios are detectable if not only the status of a system is evaluated but tendency analysis is taken into account. In machin tool industry the problem of tool-wear has been successfully tackled by logging of tool related force and torque data and monitoring the tendency of these values - thus giving an early warning when tools need to be replaced or adjusted. If this strategy is to be applied to embedded systems then the amount of data that a system needs to provide goes well beyond simply status values - embedded GNU/Linux allows to monitor systems at runtime down to the kernel internals and provides a multi-level logging facility that is the basis of any tendency-analysis system. Making this data available to off-line systems is trivial and possible with low resource requirements. Off-site logging allows to perform tendency analysis not only over long terms but allows to detect correlations between events on dif-

ferent devices and frees such analysis of the resource constraints that apply to the embedded system itself. The potentials for early fault detection and maintenance response has, I belive, not been appropriately considered by embedded OS/RTOS developers.

25.2.2 Network Capabilities

High end embedded Systems are not only required to offer remote administration in many cases, but in addition the demand for system update and system independent remote monitoring is moving into the list of mandatory features. Linux and also embedded Linux offer many possibilities to satisfy these needs at a high level of efficiency flexibility and security, at the same time extending network related feature far beyond common demands.

Network resources

One of the strengths of GNU/Linux is its network capabilities. These include not only a wide support for protocols and networking hardware, but also a wide variety of servers and clients to communicate via network links. Naturally, a system that provides a large number of network resources also needs to provide appropriate security mechanisms to protect against unauthorized access, data leakage and DOS (Denial Of Service) attacks. In this respect Linux has evolved very far - especially the latest 2.4.X kernels provide a highly configurable kernel with respect to network access control and network logging.

Remote Administration

Reducing costs is a primary goal of much of the technical development effort being done. A major cost factor in embedded systems is long term maintenance costs. Not only the direct costs of servicing the devices on a routine basis, but also the indirect maintenance related costs of system down-times and system upgrades are an important factor. A reduction of these costs can be achieved if embedded systems have the ability of remote administration. This encompasses the following basic tasks:

- remote monitoring of system status (local shell access, web-interface, off-site logging to a central facility, etc.).
- remote access to the system in a secure manner allowing full system access. This can be done via encrypted connections (VPN) allowing a high level of security for almost any protocol used between the target system and a central management-system.
- the ability of the system to contact administration/service personnel via mail/phone, based on well definable criteria.

- System tendency analysis - allowing early fault detection and intervention as well as post-mortem analysis.
- upgrade-ability of the system in a safe manner over the network, allowing not only full upgrades but also fixing of individual packages/services.

A GNU/Linux based embedded system is well suited for these tasks, providing well tested server and clients for encrypted connections, embeddable webservers as well as system log facilities that are capable of remote logging and inter-operation with almost any Server-OS. Outgoing calls from an embedded system, that are necessary to satisfy these criteria are also well established in GNU/Linux, allowing for connections to be established via any of the common network types available, including dialing out via a modem line.

A missing capability of linux to date is a light waigt rstatd implementation. Current rstatd utilizes the `proc` interface which is too heavy waight (too many system calls to access data *note1) and is not realy that suited for judging the helth of an embedded system, suggestions to improve monitoring capabilities via a centralized monitoring server have been suggested ??upermon, but although this concept is well suited it needs adaptations to the specifics of a given environment to be efficient (i.e. what valuae to monitor, intervall of monitoring). Ine specific problem of monitoring embedded systems is that data needs to be buffered as conections may not be permanennt and/or monitoring frequency would need to be too high to detect all relevant developments, buffering of data aswell as preprocessing on the embedded nodes can improve monitoring verbosity a lot, and improve detection of problems far befor they become fatal (i.e. temperatur increas, OOM conditions etc.).

note1: it is posible to build efficient `/proc` interfaces if cirtain provisions are taken, see proc_utils project ??roc_utils

Scanning the Potential

The last section listed a number of tasks that a remote administatable system should be able to perform, but this is definetly not the full suite of offerings a GNU/Linux system will have in the network area. The degree of autonomy of an embedded system can be pushed up to that of a server system - allowing for dialin support for proprietary protocols to fit into a non-unix environment smothely. NFS, the network filesystem, can not only be incorporated as a client in an embedded system, but also as a server, allowing for a central server or administration system to mount the embedded system for monitoring and upgrade purposes. This way giving virtually unlimited access to an embedded system over the network. At the same time all of these services can be provided in a secure maner by running them over VPN's or encrypted lines. This capability of 'stacking' services is one of the strengths of GNU/Linux networking - and again, you don't need to rely on a specialized software package, you can rely on well-tested and widely deployed setups that will give you a maximum of

security paired with a unprecedented interoperability with other OS's, protocols and network-media.

25.3 Security Issues

My personal belief is that not so much power consumption or processing speed but security will be the key issue in embedded systems in the near future. Reliability was one of the demands from the very beginning on - security, on the other hand, has been neglected. The more embedded systems become complex, offer extensive user intervention and utilize the ability to interact with local networks and the Internet, the more security related issues are emerging.

25.3.1 Linux Security

GNU/Linux for servers and desk-top is well suited for sensitive computer systems. Its security mechanisms are challanged on a daily basis from script kiddies and 'professional' hackers. Although this is not a very pleasant way of getting your system tested, it is a very efficient way. A system that is deployed in a few hundred to maybe a thousand devices will hardly be tested as extensively as the GNU/Linux system. This means that an embedded Linux or realtime Linux system is relying on the same mechanisms that are being used in servers and desk-top systems. This high degree of testing and, at the same time, the full transparance of the mechanisms in use, due to source code availability, make a GNU/Linux system well-suited for systems with high security demands.

Standard services that a linux system can provide:

- Firewalling and network filtering capabilities
- kernel based and user-space intrusion detection
- kernel level fine graen capabilities allowing for precise access control to system resources
- user level permissions and strong password protection
- secure network services
- well configurable system logging facilities

These posibilities taken together allow not only monitoring systems with respect to current actions taking place and intervening if theses are inapropriate, but also for detection of system tendancies and response to developments far before failure occurs. This tendency monitoring covers hardware (e.g. temperature detection or system RAM testing) as well as monitoring system parameters like free RAM, free discspace or timing parameters within the system (e.g. network responnse time to ICMP package). A vast majority of the hardware related

failures are not abrupt, but develop slowly and are on priciple detectable - having an embedded OS/RTOS that can provide this service can improve the system reliability aswell as the systems security.

25.3.2 Talking to devices

Most embedded systems will have some sort of specialized device that they are talking to, to perform the main system task - may this be a data-acquisition card or a stepper motor controller. These 'custom devices' are a crucial point in the embedded Linux area, as these will rarely rely on widely deployed drivers and have a limited test-budget available. So to ensure the overall system security, a few simple rules need to be kept in mind when designing such drivers and the advantages of releaseing a driver under an open license like the GPL should be concidered for such projects as this increases the test-base.

Regular linux device drivers operate in kernel space. They add functionality to the Linux kernel either as builtin drivers or as kernel modules - in either case there is no protection between your driver and the rest of the Linux kernel. In fact kernel modules are not really distinct entities once they are loaded, as they behave no differently than built-in driver-functions, the only difference being the initialization at runtime. This makes it clear why device drivers are security relevant: a badly designed kernel module can degrade system performance all the way down to a rock-solid lock-up of the system. A really badly designed driver will not even give you a hint at what it was up to when it crashed. So drivers, especially custom drivers, must aim at being as transparent as possible. To achieve this, a flexible system logging should be anticipated. This may be done via standard syslog features as well as via the /proc interface and ioctl functions to query status of devices. The later also can be used to turn on debugging output during operations, a capability that, if well designed, can reduce trouble-shooting to a single email or phone call. Aside from these logging and debugging capabilities, a driver design must take into account that there is no direct boundary between the driver and the rest of the kernel. That means the driver must do sanity checks on any commands it receives and in some cases on the data it is processing. These checks not only need to cover values/order and type of arguments passed, but also check on who is issueing these commands - the simple read-write-execute for user-group-other mechanism of file permissions is rarely enough for this task.

RTAI and RTLinux devices are not that much different from regular linux devices with respect to security considerations, but they differ enough that this difference should be mentioned explicitly. Noting this for RTLinux and RTAI only is only due to the fact that oure work covers this RTOS variant of Linux, but basically it should hold true for the other flavours of realtime enhanced Linux variants (corrections appreciated).

A simple example of setting up a secure RTLinux device would be a motor controller kernel module. This module must be loaded by a privileged user (the

root user) and needs to be controlled during operation. To achieve this:

- Load the module and system boot via init script or inittab.
- change the permissions of a comand FIFO (dev/rtfN) to allow a non-privileged user to access it.
- send a start/stop/controll command via this FIFO as the unprivileged user.
- check the validity of the command and its arguments.
- log such events with timestamps and user/connection related information to the systems log facility.
- monitor the logged events and follow development of driver parameters during operation.
- document the system behavior in a way that deviation can be located in debug and log output.

Note that you also can use the /proc filesystem interface for starting and stoping of rt-threads in kernel space, or utilize the standard complient sysctl vacilities.

If a scheme of this type is followed, then operating a system with custom devices will exhibit a fair level of security. Clearly, a non-standard device will also require an increased amount of documentation and instructions for the operator, as the behavior of non-standard devices can hardly be expected to be well-known even to knoledgable administrators. (TODO: monitoring facilities)

25.3.3 Kernel Capabilities

A feature of the linux kernel that is slowly finding its way into device drivers and into applications is its ability to perform permission checks on requests at a more fine-grain level that the virtual filesystem layer (VFS) can. Kernel capabilities are not limited to the normal filesystem permissions of read-write-execute for owner-group-others. Resorting to these capabilities in the kernel, allows controlling actions of the driver, such as introducing restriction on chown or releasing some restrictions like on ID checks when sending signals (which allows unprivileged users to send signals instead of making the entire process a privileged process). These capabilities require a cleanly designed security policy for the drivers. The name of this kernel feature says it very clearly: it's control of capabilities not a security enhancement as such. No system is secure or insecure, but some systems can be configured to be secure and others simply can't. The goal of any implementation using kernel capabilities for access control should be to replace global access settings by resource specific access restrictions. By

this means, one can prevent the root user from accessing the device altogether as well as give an otherwise completely unprivileged user full access to a specific resource.

A often neglected resource in the Linux kernel is the proc filesystem, asside from the obvious write access problems, that is if write access to files in /proc is granted then systems must due sanity checks on passed values, there also is a risk with read access granted to non-privileged or generally operational personell. This risk stems from the information in the /proc filesystem that can reveal internals of the kernel that might not otherwise be visible and thus allow atacking the system with a high ïnsider-know-how:̈ Files to mention in this category are the kcore file, and the entire trees of system settings below /proc/ in fs, net, sys etc. As an example what is ment here take /etc/exports a file listing all hosts that may NFS-mount a local file, this file typically is set readable by the root-user only but is m̈irrord̈in by a simple `cat /proc/fs/nfs/exports` which obviously bypasses the intention of the access permision of /etc/exports.

Not to lead to a mistake - this is not claiming that the proc filesystem is bad in principal - it is just explicidly mandating that it be taken into account for when designing the security policy and the access model of an embedded system. And in some cases it may be sensible to trim down access rights in /proc or even removing some files completly.

25.3.4 Network integration

By now any reader will belive that embedded GNU/Linux is only for paranoia struk developers - if not - then the GNU/Linux capabilities and efforts in the network security area are going to convice you. As networking was a strong-point in Linux from the beginning on security issues emerged early. To these security issues the move towards the IPv6 infrastructure has come in recently. Both subjects are highly relevant for embedded and distributed embedded systems. As it is not posible to even only list all the netowrk and IPv6 related works on-going, only a few pointers should be given here.

- IPv6 support - since the early 2.4.X kernels IPv6 is being supported in the latest 2.4.X kernels it can be called fully supported.
- IProute2 - kernel based policy routing. This naturally coverst stnadards like sourcs/destination based routing polidy but in the Linux kernel this has been extended to allowing TOS or even UID based routing and queuing policys.
- QOS - this has been around quite a while, it's full power is emerging in recent kernels with new policy concepts like HTB (Hierachical Token Bucket) reaching production quality.

A comon problem in embedded systems is that customers will request notoriously insecure protocols to be available (like telnet or SNMP), the solution

tha embedded GNU/Linux can offer here is to allow every insecure, clear-text äuthentication ÿou would like and pack it up in a secure encrypted tunnel. This not only has the advantage of making insecure protocols secure, provided access can be limited to a trusted host, but it reduces the the design demands conciderably if one needs only to take the VPN into account and not every posible protocol. It should though be noted that this does not handle the problem of Denial Of Service (DOS) atacks against such systems.

Integrating embedded systems in existing netwrok environments opens a new set of problems that need concideration. Many system services relie on each other and this can lead to irritating servic/protocol interdependancies. As an example take the system command `route`, if DNS is blocked then this comand will hang until it reaches the timeouts for every DNS request in the list, and that can be quite long - inaceptably long when called from some system script. To set up a system in a secure maner requires that such dependancies be analized or at least tested.

One posible strategy for this problem is to let the embedded system perform all such operations in a r̆awm̈ode and only resolve data for analysis off-line. Not taking these effects into account can lead to systems going to extreem load-averages if a remote service fails, so basically any local service that relies on remote servers must have some exit strategy to ensure that it will not bring the system to it's knees.

25.3.5 Boot loader

Putting boot-loaders into a separate section about security is due to the experience with many systems offering insecure setups right at the boot prompt. A substantial number of the LILO based system encountered allows for passing a simple `init=/bin/bash` at the `LILO:` prompt and a root-shell with no restrictions was on the screen...

It must be clear to system developers that the handy boot-loader prompt during development is a serious risk during operation and that a security policy should always include a clear statement on the acceptable boot selection and boot commandline access. And access to a certain extent can be restricted by not compiling in any not required resources into the kernel - typically NFS should only be compiled into the kernel if the system is to operate permanently as a NFS-root based system. Making NFS-root available on systems that actually don't need it allows for full access by providing an NFS-server of the hackers-choice. The same naturally holds true for quite a few other Linux kernel options. So to repeat, the kernels capabilities and the boot-loaders capabilities need to be part of a serious security policy for an embedded GNU/Linux system.

If, for what reason ever, a boot-loader is selected that gives full access to the system, then it must be made clear to the customer that physical access to the device is a threat to security. Preferably such systems should at least record the commandline arguments passed for later reconstruction of system problems.

25.4 Resource Allocation

Embedded systems, even in their high-end variants, are resource constraint systems by desk-top or server standards. In embedded systems resource allocation need to take the overall system resources and operation modes into account. Resource constraint systems need strategies to reduce demands to a minimum and at the same time include methods to temporarily increas or dedicate a large amount of these resources to critical actions - resources in question being not only mass-storage media RAM and CPU consumption, but also time, system-response and network capabilities.

25.4.1 Time

Complexity of operations require many optimization strategies that were designed for server and desktop systems to be utilized in embedded systems as well. As standard GNU/Linux targets interactive usage and optimized average response, some of these strategies are not ideal for embedded systems. Considerations for more predictable timing and well-defined system response to critical tasks is necessary. In this respect the nogoing enhancments in the 2.6.X traks of kernels is of great interest, although this development is targeting scalability, developement of fine-grain synchronisation and kernel-preemption is of great interest for embedded systems aswell (see part 2 Preemtive kernel)

Standard Linux

Linux has a record of squeezing a lot of performance out of little or old hardware. This is done by relying extensively on strategies that will favor interactive over non-interactive events. For instance, writes to disc can be delayed substantially and Linux will buffer data and reorder it, writing it in a continuous manner with respect to the discs location, and out of order from the user's standpoint. These and other strategies are well-suited to improve average performance but can potentially introduce substantial delays to a specific tasks execution. This is to say that peak delays of a second or even more can occur in GNU/Linux without this indicating any faulty behavior. As embedded systems are generally resource constraint systems such optimization strategies are an improvement in most cases, but increasing system complexity and the potential of a networked system reaching very hight loads (just imagine a network on which many other probably faster systems are broadcasting all kinds of important server announcements...) can degrade the system's response to high priority events dramatically. This is to say that an embedded GNU/Linux system better not have any timing constraints at all and should not rely on the system's catching a specific event. If there are no such constraints with respect to timing, then an embedded system running a scaled down standard GNU/Linux will well suit most purposes and operate very efficiently.

Soft Realtime

There are many definitions floating around what soft-realtime is. I'm not an authority on this section, but give the definition used here to prevent any misunderstandings. Under Soft-realtime a system is capable of responding to a certain class of events with a certain statistical probability and an average delay. There is, however, no guarantee of handling every event, nor is there any guarantee for a maximum worst case delay in the system. In this sense every system is a soft realtime system. Of course, the term is used for systems that have enhanced capabilities in this area. In most cases this will mean:

- high-resolution timers
- a high-probability of reacting to a specific class of events. High probability in this sense means 'higher than regular Linux'.
- low average latency, again low relative to regular Linux.

Soft realtime systems are well-suited for cases where quality depends on average response time and delays, like video-conference and sound processing systems, and if the system will not fail or get into a critical state if the one or other event is lost or delayed strongly. Simply speaking, soft-realtime will improve quality of time related processing problems, but will give you no guarantee. So you can't have safety critical events depend on a soft-realtime system. There are multiple implementations of soft-realtime for Linux, starting out at simply running a thread under the `SCHED_FIFO` or `SCHED_RR` scheduling policy in standard Linux all the way to the low-latency kernel patches that make the Linux kernel partially preemptive (please no flames...thanks). Soft Realtime variants of Linux include RED Linux, KURT, RK-Linux and the low-latency patch of Ingo Molnar.

The current development tree of the main-stream Linux kernel, 2.5.X includes the preemtive kernel patch in it by default, and thus is a soft-realtime kernel capable of satisfying many timing demands in standard Linux that were requireing soft-realtime variants up to now. At time of writing the preemtive extension made it all the way up to 2.5.18 so I guess it will not be kicked out again ;)

Hard Realtime

There are many systems that obviously have hard-realtime requirements, such as control or data-acquisition systems. But there also are a large number of systems that don't have quite so obvious hard-realtime demands: those systems that need to react to special events in a defined small time interval. These systems may be performing non-timecritical tasks in general, but emergency shutdown routines must still be serviced with a very small delay independent of the current machine state. In such cases a hard-realtime system is required to guarantee

that no such critical event will ever be missed, even if the system goes up to an enormous system load or a user-space application blocks altogether. The criteria for requiring hard-realtime as opposed to soft-realtime are the following:

- No event of a specific category may be missed under any circumstances (e.g. emergency shutdown procedure)
- the system should have low latency in response to a specific type of event.
- periodic events should be generated with a worst case deviation guaranteed.

Note that these three criteria do overlap in a certain respect and could be reduced to a single one, that being to guarantee worst case timing variance of a specific event class, but that's not what I would call a self-explanatory definition. A hard realtime system naturally also will provide high-resolution timers andappropriate alarm/sheduling functions.

RTLinux, and a non-POSIX derivative of it, RTAI/RTHAL, aswell as RTAI/ADEOS fall into the class of hard-realtime Linux variants (if you know of any others let me know). These are based on three principles (that are covered by US Patent 5,995,745)

- Unconditional Interrupt interception.
- delivery of non-realtime interrupts to the general-purpose OS as soft-interrupts.
- Run the general-purpose OS as the idle task of the RTOS.

By providing communication mechanisms that allow data exchange between RT and non-RT tasks via shared memory RT-FIFO and POSIX signals, as well as extending RT-execution to allow for user-space realtime in RTLinuxpro, a full integration of demanding realtime tasks into an embedded Linux based system is achievable. This extends the RTOS to include the full feature of GNU/Linux without limits.

A featur that is available in the latest versions of RTLinuxpro is to reserve a minimum CPU time for non-real-time (that is Linux) to ensure that no rt-task can actually monopolize the systems resources and thus de-facto crash the system (that is it will not crash it will only freez if a rt-tasks uses 100% of the CPU time continuously - for all practical purposes and from a user perspective the box is rock solid locked). The ability to reserve CPU time for Linux is relevant for systems that need to report such errornous behavior and may not simply fail silently.

25.4.2 Storage

Embedded GNU/Linux systems can take advantage of commodity components which can be an interesting opportunity for some classes of embedded systems - for the majority the standard PC storage media are not usable. Typical embedded systems will require access to solid state media as mass storage devices NVRAM SDRAMs and Flash-Memory devices - The MTD (Memory Technology Devices) project has expanded the spectrum of devices into this class of storage devices. At the same time doing this in a way that is highly compatible to procedures developers are used to from desk-top PC's - this simplifies migration and development substantially. The Second class of storage media that is of interest to embedded devices - although not specific to these systems, are network-storage media.

Memory subsystem

If one compiles a curent Linux kernel one might easaly think it is not well suited for embedded systems - the linux kernel size has grown substantially between 2.0.X and 2.2.X and again between 2.2.X and 2.4.x. This has moved the minimum memmory demands up to 4MB weras a 2.0.X kernel could confortably operate with 2MB RAM. So is a 2.4.X based system not usable for embedded linux ? Not only did a rich and interesting set of featurs get added in the 2.4.X kernel series, notably the clean integration of MTD (Memory TEchnology Devices) and iproute2/QOS but also the way the kernel manages memmory resources has improved substantially and that is why even for resource constraint systems a 2.4.X kernel will performe better than a slim 2.0.X kernel. Major improvements are in the buffering mechanism the cleanup of cache-allignment and direct access to peripheral buffers from userspace (kobuf) and other low level extensions. It's not posible to describe the full memmory subsystem in a few sentances - the simple message is 2.4.X kernels will manage memmory resources on a low-memmory system better than a 2.0.X/2.2.X and the increase in kernel size is well worth it. Aside from performance issues the memomory management of the 2.4.X kernels also exhibit better security characteristics than early kernels.

Mass storage

Storage media used in embedded GNU/Linux systems are simply standard PC devices in some cases, that is, normal hard disks and PC memory. In typical embedded GNU/Linux systems though one will find dedicated storage devices, like DOC, CF, DOM, NVRAM or flash devices. Aside from these devices requiring special system behavior (i.e. wear-leveling for CF disks) optimized filesystems and boot-strategies are available for embedded systems.

Offsite resources

The term network-storage will obviously be associated with NFS or SMB partitions/network-drives, beyond these GNU/Linux includes a number of other off-site storage media accessible via the network. The range here is from advanced distributed file-systems like coda (that offers some advanced security and operation features but functionally is comparable to NFS) all the way to Network block device support in the kernel that allows accessing a mass-storage media like a hard-disk over the network like a regular local device. Aside from these clean solutions any automatable file transfer protocol can be 'misused' to keep files off-site and load them on demand. The goal of all such efforts is to allow for temporarily increasing local resources to the system. Remote mass-storage media can not only increase the local mass-storage resources but can even be used to increase the virtual memory available locally (that is you can swap over a network block device).

The minimum list of libraries

Some of the library problems were mentioned above, glibc is a very large and powerful library, but for minimum systems it's a problem since it is very resource consuming. Nevertheless we stick with glibc, because reducing its size is not only complicated (you must figure out all function calls that are unused and remove them), but also because it poses a compatibility problem. If you try to optimize by modifying libraries, you lose compatibility with your desktop system. At the same time, it means maintaining a private version of the lib, and you don't want to maintain your own libc track!

Stripped libraries are dramatically smaller, and since debugging can comfortably be done on the desktop-system, there is no need to include debug symbols on MiniRTL. The same holds for executables that can be stripped, thereby massively reducing size. To reduce the number of required libraries, it is best to define a set of libraries for the minimum system and then strictly build on those. This is not such a big problem, due to the vast amount of software/sources on the Internet, it is quite easy to find editors, scripting languages and the like that will not need any special libraries. Naturally, the system will have a little bit of an archaic touch, but that's ok; you're not expected to work full time with ash and ae as your shell and editor. For administrative jobs, you can get used to it.

for glibc-2.0.7pre6 assuming network support the minimum set of libraries is:

```
ld-2.0.7.so
libc-2.0.7.so
libcrypt-2.0.7.so
libdl-2.0.7.so
libncurses.so.4
libnsl-2.0.7.so
libnss_db-2.0.7.so
libnss_dns-2.0.7.so
libnss_files-2.0.7.so
libresolv-2.0.7.so
libss.so.2.0
libutil-2.0.7.so
libuuid.so.1.1
```

libc - which one

Which libc ?

This is a heatedly debated issue and the answer is not simple. my personal approach is to check what feature I really need and go for the library that can satisfy these needs even if its not the newest and hottest from gnu.org. So for embedded systems glibc-2.0.X compiled with 2.95.X gcc has delivered the best results for me. An issue that needs to be checked when compiling a glibc for an embedded system is if one really needs the threads extension - many systems don't need it and the glibc is quite a lot smaller without linuxthreads.

As an example here is glibc-2.0.7 compiled with gcc-2.95.2:

```
-rw-r--r--   706681 Nov 12  1999 libc-2.0.7.so
-rw-r--r-x   639032 May 23 22:11 libc-2.0.7.so
```

A number of dedicated embedded libc variants have been emerging, dietlibc as one of the more prominent. Basically this may be a solution, but the advantage of a smaller libc is payed for by the loss of compatibility with the desk-top and the problems one encounters compiling packages that compile out of the box with glibc. That is not to say that these projects don't work, it is to state that these specialized libc variants are excellent for very small and restricted systems that will live with busybox + tinylogin + ash and some user apps from the project, once you get into the range where you need libssl or openssh or want to cross-compile for other platforms, the gain of a reduced libc is marginal against the handling issues. So as a rule of thumb, for systems with a very reduced user-space these reduced libc's are fine, for a full-fledged GNU/Linux system I doubt they are a good solution.

A note on library optimizers, there are a number of them around, they will reduce glibc quite a bit, but generally the same rule holds true, the gain is noticeable on very small system on larger systems, especially those where libc.so is not the main chunk on the system the gain of these optimizers is drastically

reduced. And an issue that is easily overlooked: You need to do a security assessment of these libs if you need to guarantee security on your embedded platform. Glibc is not bug-free ma by, but the number of users testing it every day locates bugs relatively reliably.

Feedback - especially on this section - would be very appreciated !

25.4.3 Network

Network resources have been mentioned a few times all ready - in this paragraph we focus on resources in distributed embedded systems. One could state the distributed embedded systems are actually characterized by the ability of resources and not only data-items to change there locality.

Centralized Services

To integrate embedded systems and distributed embedded applications into an existing network a key requirement is that such an embedded OS/RTOS be able to communicate based on standard protocols. This clearly is one of the strong points of GNU/Linux as it supports a very large number of standard protocols and allows to move such high-level services to central authorities with little effort. Standard services supported as client and server include BOOTP, DHCP, DNS, SNMP (who put the 'Simple' into SNMP ??), SMTP, FTP, HTTP etc. etc. which results in a high level of inter-operability with existing network infrastructures. Aside from this being a requirement for inter-operability this naturally allows a further shift of resource from local media to centralized servers. This not only reduces local resource demands but also increases the available data-pool for early error detection as well as simplifying administration and maintenance for distributed systems.

iproute2/ipfilters

Especially distributed embedded systems have limited bandwidth available for communication with central services and logging facilities. Resource limitations need not only be slow media like a 28K analog modem, but on a 486 based SBC it is hardly desirable that a 100Mbit link ever deliver packets at the full speed as this could simply bind too much CPU power for some systems.This limited resource requires the ability to allocate resources to critical communication tasks and at the same time prevent any task from monopolizing the available bandwidth. As distributed systems are becoming more complex simple source destination based policy routing will not due. Current linux kernels include capabilities to allocate bandwidth to specific protocols/uid's/TOS etc. allowing to ensure that an interactive administrator session over a slow link will not be de-facto blocked by a 'syslog-burst'. Aside from these allocation capabilities this naturally also improves security due to limits for potential DOS ports/protocols

and with ipfilters fine grain filtering on the network layer is possible, again being an important security aspect.

rt-networking

Recent development in Real Time Linux have focused on extending the realtime capabilities beyond the single node UP and SMP system to distributed RT-systems that utilize commodity component computers network hardware (ehternet and firewire). These realtime networking efforts have now been merged into the embedded Real Time Linux allowing to extent realtime constraints over networking infrastructure. Distributing computational resources, which is very demanding on the networking layer, can be extended with realtime capabilities becoming available. Basically this allows to reduce the locality of resources which improves the flexibility of embedded systems and opens new possibilities in embedded system design. With realtime networking available for embedded Real Time Linux it is possible to tightly synchronize nodes and offer statically allocated channels between embedded nodes pushing QOS effort all the way to hard-realtime. Current implementations are still limited with respect to security provisions (no encryption/authentication for realtime networking in any of the available hard-realtime implementations) but conceptual work in this are is under way (ref fsmlabs security initiative). It might be noted here that the question of security in realtime networks has generally been neglected and (all ?) implementations simply assume they will operate in a 'secure' environment.

Currently available implementations for Ethernet:

- RT-Net (for RTAI and older RTLinux versions)
- LNET (for RTLinux/Pro only)
- lwIP RTLinux/GPL (extension for RTAI in concideration)
- RTSock RTLinux/GPL (it is though fairly version independant, thus making it available for other RT-variants would be littl effort)

FSMLabs LNET also support IEEE 1394 Firewire (A/B)

Furthermore RT-CAN based on the CanOpen project is available for RTAI and RTLinux/GPL.

for details see (part3 RT-Network implementations, Part 7(?) RT-Networks selection Guide).

25.4.4 Filesystem selection

The descision which filesystem to use is not easaly answered. There are differenct aspects to take into account.

- access modes - read/write, read-only, access to images

- storage bandwidth - fragmentation, locality of files, compressed read/write on slow devices
- storage density - fragmentation, superblcok-copies, filename-length.
- security issues - mount options, file types supported, fault tollerance.
- operational handling - creation, mount performance, recovery options.
- hadware issues - access performance, waer leveling, scalability.

In the folowing paragraphs only a subset of the filesystems available in GNU/Linux is covered, not all are suitable for typical embedded setups - but if none of the filesystems mentioned here offer what you might require then give the documentation in the Linux kernel tree a look for other options.

Boot FS

The boot file system is tightly coupled to the operational mode the system will be in during boot up. The selection of the filesystem is not only performance related but one must take procedural issues into account, if the filesystem needs to be modified by a customer then a filesystem like msdos might be preferable as its easier to manipulate with common desk-top OS's on the other hand if the system is a b̈lack-boxẗo the customer, then performance and security issues can be put at the top of the demand list.

The options available are read-only filesystems, read-write filesystems, both as runtime and/or dedicated boot filesystems. Naturally one can go for a raw medium with a compressed filesystem image - this will give you the smallest possible boot-image size and thus the least storage demands but will result in a had to manipulate and not very robust filesystem with respect to media errors.

Read-ONLY root-fs: romfs, cramfs

- romfs: romfs is uncompressed and so does not need to decompress.
- cramfs: is compressed and indexed, and so has shorter boot time
- jffs2: this is actually a read/write fs but as boot medium jffs2 is supported read-only by some boot-loaders (ppcboot).

Readonly filesystems have a clear advantage - you can't modify them at runtime even if you gain root-permissions. They have a just as clear disadvantage, if you want to update the filesystem the entire filesystem must be replace, if something goes wrong you might loss access to the device completely and if your embedded GNU/Linux system is on a satellite 64000k above the equator... even if it's a bit closer than that updating read-only filesystems is a problem for any device you don't have physical access to.

Read-Write root-fs:

- jffs2: in read/write mode this compressing filesystem can yield the smallest RAM/Flash media-requirements.
- msdos: not a very elegant solution but for some boot-media this is a simple way of getting a GNU/Linux system to boot
- minix: this is a old UNIX filesystem, supported from the very first linux kernel versions

Jffs2 is compressed but must do a full scan on boot up which results in a somewhat slow mount operation, generally this is only of concern for systems that have to provide extremely short boot-times - so if this requirement is given then a compressed filesystem is probably not the best solution (assuming that a system requiring fast boot up will not have a slow boot-media, which would off course profit from a compressed filesystem). It can be read/write mounted after booting , at system boot it generally will be accessed read-only . Support for jffs2 as boot-filesystem is moving into the boot-loaders slowly (grub ppcboot), but currently booting of jffs2 is not straight-forward. The obvious advantage is the flexibility of a read/writable filesystem and the efficiency due to compression. One point that applies to other boot-filesystems as well is that it is non-trivial to access and modify boot-file-images based on jffs2, at least not for the untrained personnel.

Using an msdos filesystem as boot-media is a simple way to get some devices to work with Linux especially if the boot-medium comes with a DOS-filesystem pre-formated and the intelligent BIOS will not boot from anything else. A further advantage is that it is supported by many OS's so if the filesystem resides on a removable medium and requiring a Linux desk-top for manipulation is not acceptable to the company, then an msdos file-system can be a solution. As long as you don't write to it during a power-loss msdos is quite robust, and it is quite efficient with respect to the overhead the filesystem will require. The most notorious limitation of msdos though, is the well know 8+3 name-length limitation and the case insensitive behavior. For these reasons msdos boot-fs is to be considered a last-option in my opinion.

As noted above for embedded systems minix also is used quite frequently, this actually is a UNIX filesystem, it is though limited with respect to the supported file-name length (30 characters) and the maximum directory depth, the later generally is not a problem for small embedded systems, 30 character name-length restriction is an issue, as minix will silently truncate filename length so this potentially can lead to hard to locate problems. Minix is relatively efficient with device usage, especially if it is on a boot-device with very few files limiting the number of inodes at filesystem creation to the actually required number can optimize media usage quite well.

standard Linux FS: ext2

For embedded systems that have plenty of disk-space available an ext2 file-system is fine, it's fairly robust and most linux users are acquainted with using it, so there are little handling issues involved with ext2. One problem with ext2 as a boot file-system is that if it ever becomes inconsistent it requires user intervention at the console (typically if you do a few power-fail sessions in a row...) and it also will mandate fs-checks to be run after N reboots (N being somewhere between 10 and 20 commonly , but that's a configurable parameter). The wide use of ext2 is also due to some boot-loaders (notably LILO) not being able to boot of journaling filesystems like reiserfs or jfs directly so commonly systems have a ext2 boot-partition even if running of some other file-system during regular operation.

journaling fs: reiserfs, ext3, jfs, jffs, jffs2

For all embedded systems that must prevent requiring the ability of direct user interaction at the console should use a file-system that is fail safe against power-loss even while writing. This class of filesystems has been becoming available on linux systems within the past year or so and has now matured to a point where it actually is ready for production systems. For embedded systems with large storage media, like hard-disks ext3, reiserfs or jfs can be of interest, the later two requiring a minimum filesystem size of 16 MB. Ext3 being a ädd-onto ext2 will run on smaller media as well but will not be very resource efficient on small media.

As jffs/jffs2 is covered in a bit more detail below, so here it should just be stated here that it is a journaling file-system that is quite robust to power-fail situations and jffs2 is very space efficient due to its compression. Thus for small media (¡4MB) jffs2 is probably the only real option out there in the embedded Real Time Linux world. A boot_media size, for an embedded system with local filesystem (that is non-nfs-root setup), of 2MB can be considered a hard-limit.

Storage efficiency

All comparisons of filesystem efficiency will vary with the actual filesystem used, the sizes of the individual fils etc. aswell as capabilities of filesystems like compression journaling and the like. To give a rough guidance to the efficiency of a filesystem a fully operational embedded filesystem (MiniRTL V3.0) was taken as a basis, as this filesystem provides the most commonly requested services and a fairly complete user-land, note though that this filesystem does not contain any X related applications and libs, it is thus a typical deeply embedded filesystem for embedded realtime systems. The file-type distribution of the MiniRTL filesystem

	Type		
	symbolic links	187	
	regular files	236	
	device special files	80	
	directories	57	

Comparison of Data storage efficiency of different filesystems with compreseed and uncompressed tar archive (.tar .tar.gz .tar.bz2) not taking filesystem overhead (journal managment, superblock copies etc) into account. The difference between filesystems is due to internal fragmentation and padding of the files.

	Bytes Used	Type	Media Usage	
	1010529	minirtl_fs.tar.bz2	30.42%	
	1092819	minirtl_fs.tar.gz	32.89%	
	1277952	minirtl_fs.cramfs.img	38.47%	
	1409368	minirtl_fs.jffs2.img	42.42%	
	2756540	minirtl_fs.jffs.img	82.98%	
	2950144	minirtl_fs.ext2	88.81%	
	2950144	minirtl_fs.ext3	88.81%	
	2960384	minirtl_fs.minix	89.11%	
	3061760	minirtl_fs.tar	92.17%	
	3321856	minirtl_fs.reiserfs	100.00%	

`Note: cramfs and jffs2 are compressing filesystems`

The second comparison takes the effectiv filesystem overhead into account - that is what amount of a storage device is actually available if the media has a raw size of 4096 KB.

	Type	FS-Size	Meta Data	Nete Size	effi-ciency	
	msdos	4072	0	4072	99.41%	
	jffs (NAND)	4096	48	4048	98.82%	
	minix	4049	1	4048	98.82%	
	jffs2 (NAND)	4096	160	3936	96.10%	
	jffs (NOR)	4096	192	3904	95.31%	
	ext2	3963	1	3758	91.74%	
	jffs2 (NOR)	4096	640	3456	84.60%	
	ext3	3963	1043	2716	66.30%	

(note: jffs theorectically has no limitation on what garbage collect might need)

Reiserfs and jfs was not taken into account as you can't use it on a 4MB medium any way, the overhead of jffs and jffs2 is hard to account for - documentation for jffs2 state that the garbage collection overhead is 5 erase blocks

amounting to 48K for NAND and 640K for NOR Flash (for typical block sizes of 8K and 128K respectively). The available jornaling filesystems (reiserfs,jfs,xfs) and the ext3 filesystem (a kind of jornaling filesystem extension to ext2) are not suited for very small footprint embedded systems but are well suited for 32MB++ filesystems, note though that the jornaling concepts of reiserfs and ext3 do not protect against data loss on power failure, they only guarantee a consistant state (which may be fairly old). Generally embedded systems will require some form of NVRAM to store critical data (status, operational state, error infos) to be retrieved after a system failure or power-cut, NVRAMs in most cases don't utilize the capabilities of a filesystem but are simply treated as a contingous memory location leaving the sync to the NVRAM driver implementation not the VFS.

Last it should be noticed that most Linux file-systems have options available during formating operations that allow to optimize usage, inde numbers, superblock copies etc. for embedded systems it pays of to give these options a close look. At the same time it must be warned that playing carlessly with such options can result in loss of compatibility (for instance restricting the name-length to 14 chars in minixfs would probably break things quite frequently) and can touch security oissues aswell. As an example one can consider optimizing a file-system by limiting the number of inodes created or setting the reserved disk-space for root to 0, this should only be done if it is posible to guarantee that such settings will not result in system failurs (e.g. on read-only used filesystems this should be safe).

To get the real size requirements of the media one needs to take both factors, compression (or storage efficiency) and filesystem overhead into account. Further it should be noted that for all filesystems accetp for cramfs, jffs, and jffs2 one must take an additional layer (FTL/NFTL for NOR/NAND respectively) into account which reduce the effectively available media size in the range of 5% to 10% .

It is not posible to cover all potential option/filesystem/hardware interactions here - the generall view should though come through: Linux is well suited for embedded systems at the file-system layer but one should not relie on defaults as these generally are not tuned to minimum systems or highly optimized systems but are tuned to robustness with respect to the untrained (desk-top) user, that is many options are set for reserving sotrage are for the root user and for superblock copies etc. that can not be safely ignored in standard setups. For designs that take these reduced filesystems setting into account a safe and roboust filesystem can be constructed though.

JFFS2

JFFS (Journaling Flash File System), originally developed by AXIS Communication AB, is a log structured file-system derived from LFS (ref ??). The original implementation has some limitations, a heuristic strategy for locating the last

log position which was not safe , inefficiencies with respect to the log-rotation (unmodified files got moved at a very high rate) and no support for hard-links (which is an irritation but not a real problem in most cases). JFFS also showd some instability when brutalized with power-fail cycles long enough. The basis laid in JFFS and the analysis of the deficits lead to the development of JFFS2 by RedHat Inc. (and is an ongoing project, maintained by David Woodhouse who also leads the MTD project). JFFS2 is a compressing file-system and thus most appropriate for systems with extremely small boot-media, also it has wear leveling implemented in software (log-rotation via the periodic garbage-collection task) which was optimized to reduce the necessary log-rotations by splitting the physical device into segments (JFFS operated on the entire device as one segment) which makes it well suited for NAND and NOR Flash types.

One of the important things to note about JFFS is that this file-system does not require any block-device on which to reside - it directly operates on the flash-device which not only improves the performance as there is no layer in between but also allows to take advantage of the linear address space (as opposed to block oriented devices like hard-disks) mapped to the flash-devices. This means that JFFS2 makes very good usage of the available resources as there is no fragmentation incurred due to block alignment like in traditional UNIX filesystems (e.g. ext2).

JFFS2 does have some overhead though, and this overhead for the garbage_collection it requires reserves five erase_blocks, so to give you a picture of an actual filesystem. the MiniRTL V3.0 filesystem will run in a 2048 KB device using jffs2 with an erase size of 128 KB, resulting in 640 KB reserved for garbage_collection. So although 640 KB or 2048 KB looks bad, you can actually run a jffs2 filesystem on top of mtdram.o and mtdblock.o with the neto RAM-image halved compared to a minix filesystem on a regular R̈AMDSIK.

XIP and raw media

eXecution In Place, XIP, is one of the common requests from the embedded world to Linux developers. To give a quick answer, XIP is not generally available to embedded GNU/Linux and the very few special cases where there is a solution (limited to some mips and arm aswell as ppc ... links to other platforms supporting XIP currently not known TODO: check XPI suuport) are to be considered experimental. Reasons why development of XIP in Linux is not getting off grounds, we belive, is that it is hard to find a generalized solution, and also because there is not really any need for XIP.

Common arguments for XIP are:

- requires less RAM as it executes in ROM
- reduces the amount of data moved for execution
- simplified bootstrapping as the addresses are static

- speed up of boot operation

To the first - XIP does reduce the amount of RAM required but at the price of accessing a generally slow device (ROM at least an order of magnitude slower than RAM) and the saving in RAM requires an increase of ROM as XPI does not allow the use of compressing ROM filesystems.

Second - the absolute amount of data moved from ROM to RAM will not really be reduced as it has to be read anyway and as the bottleneck of execution is the ROM access speed execution time is hardly influenced (infact execution speed will decrease in most cases on 32bit platforms). In any case where a read would be repeated (any function called twice) the advantage of the copy in RAM would prevail, and as GNU/Linux, even on small-footprint systems, utilizes the advantages of shared libraries XPI for runtime makes no sense. XIP limited to the boot process might seem plausible but de-facto only would improve RAM usage during the initialization process (which is not reused during runtime and actually feed by the kernel after the kernel-proper boot completed), so this strategy would not reduce the overall RAM demand again as the maximum RAM usage is not in the initialization code of the system.

Third - the simplification of bootstrapping is at the expense of loosing any generality of the boot concept and the complexity of modifying the setup (it's easier to let the boatloader figuer the real addresses) - generally the kernel and the applications will more often require an update than the boot-loader. And with the available boot-loaders this argument also makes littl sense for embedded GNU/Linux (see the section on boot loaders).

Principal limitations of XIP are also not to be over looked. You can't use block devices (so many of the inexpensive storage media are not available). Not all flash devices can be used for XIP, basically the problem is that you have no filesystem involved that is taking care of the wear leveling, so XIP would be limited to NOR devices, the less expensive NAND flash would not work reliably.

The only place I belive it would be justifiable for a Linux project to go for an XIP setup is if the limitations in RAM can not be over come due to existing hardware setups (commonly the case when migrating from a proprietary OS to embedded Linux). But XIP is the last option you should conceder.

And to repeat it - XIP is not faster than copying to RAM and working from there if you do it the right way, infect with a compressed fs reducing the number of bytes to actually copy from the ROM XIP may well be slower than copy, decompress and execute, even for a code block only used a single time !

So how do you get around XIP without wasting resources or raising device expenses ? Use a compressing file-system like JFFS2 or cramfs and use a bit more memory at a reduced demand of ROM (which generally results in an overall reduction of expenses).

Raw media - so that is putting the kernel/ramdisk directly on a media (like dd if=bzImage of=/dev/fd0 to drop a kernel directly to a floppy) is an option for some boot setups, but it requires that the media in question be robust on

multiple reads and does not require wear leveling (or it needs to be done in hardware). Accessing raw media without any filesystem or block device emulation in-between may be sensible but I would recommend comparing the performance of such an approach with the compressing filesystems available as the decreased data volume transfered can actually overcompensate the additional filesystem layer. Basically any storage media that maps into bus-mapping of the target board should be directly accessible to the Linux kernel and MTD actually provides some interfaces for such devices. As maintenance of such a setup can be kind of painful, a abstract, filesystem based solution sounds like the preferable solution.

Concerning the boot times of XIP systems all published comparisons de facto show that the speedup of a XIP kernel is simply the time saved by not having to decompress the kernel, this effect is not XIP related. In fact the execution times are increased and the overall system performance degrades (i.e. on a 266MHz PPC405 fork system call times as reported by lmbench **??**, increase from 4.9 milli seconds to 7.2 milli seconds). It also should be noted that a media coruption of an XIP kernel image would potentially not be detected at system start time, which is a security issue as generally a safe shutdown at system start time is posible wheras during operations this can be critical, or atleast anoying.

25.5 Operational Concepts

During the development of embedded GNU/Linux projects a few main modes of operation have evolved. These modes will be briefly described in the next sections, showing the flexibility of embedded GNU/Linux. This flexibility is a product of the wide range of hardware Linux and embedded Linux has been deployed on - ranging from commodity components embedded systems to dedicated hardware SBC's.

25.5.1 Available Boot Loaders

During the development of GNU/Linux a number of boot-loaders have been developed. Some of these are specifically targeted at boot floppies (e.g. syslinux) and some have been developed specifically for the demands of embedded GNU/Linux like the PPCBoot project for the power-pc processor family. Stemming from the arm boot loader and PPC-Boot a unified concept u-boot (microboot) has evolved lately. The significance of the boot-loader for embedded systems design is not limited to actually booting the system, additional capabilities like flashing devices or offering boot selections, support of emergency boot menus in case of a failure and boot commandlines are also essential for embedded systems - especially under the constraint of not directly having physical access to the system, features like limited filesystem access and built in device capabilities (serial lines,ethernet) become essential. At the same time these ex-

tended features madate a clean, top-down, security policy for such a device to also include security specifications of the boot-loader and the boot-process.

Generally a boot-loader installation program like `/sbin/lilo` or `/sbin/syslinux` should not be available on the embedded system other than during development, for upgrade purposes one can upload it at any time - making such a low-level tool available on-site can lead to very irritating problems if people get the idea to play around. Bootloaders like grub and ppcboot have evolved far beond simply boot-strapping a system to get it up and running, as nice as having a extensive interface for loading, debugging, and configuring the system is, one must be aware that not all of these features should be available on the deployed system. To this end grub is somewhat limited as the configuration files can be removed from the target, as grub actively reads these files during system boot (grub offers direct filesystem access features).

One thing still to concider is the size requirements for the boot-loader, for instance lilo is quite small when used with a minimum configuration (no graphics boot-menu) but will reach 200K if one work hard at it, the boot-loader resources are not dramatic for most systems but need to be considered for very small systems.

A key feature for the selection of a bootloader is the verbosity of the screens presented to the user, which generally should be assumed to have no knowledge of the underlaying OS and/or boot-process. In this respect syslinux, although otherwise quite limited, is very well suited for embedded systems as it offers multiple help-screens mapped to function keys. In other boot-loaders, especially those intended for desk-top systems (grub,lilo) the provided help is very limited, for dedicated embedded boot-loaders (PPC-Boot, miniboot, u-boot) the help facilities are extensive, but may be too complex for 'normal' users (there is a great difference between pressing ¡F1¿ and typing in ‘reginfo‘ ??-boot-commandsand then decipher the content - so clearly there is a tradeoff between verbosity to the uninitiated user and the developer, this should be considered when selecting a boot loader

As a conclusion from the above, currently u-boot seems to be the boot loader of choice supporting ,74xx_7xx,arm920t,i386,mpc5xx,mpc8260,ppc4xx,sa1100,arm720t, at91rm9200,mips,mpc824x,mpc8xx and pxa. Aside from this strong platform support it should also be noted that u-boot supports jffs2, which is to be considered the preferred boot-fs for most (all) small-footprint embedded GNU/Linux systems. As alternative, although, as stated somewhat out of date, syslinux is a reasonable selection for X86 based systems, the ‘limitation‘ of msdos filesystem support is not as bad as it sounds as this ‘primitive‘ filesystem proved to be very robust in read-only mode (common for embedded boot-fs). Finally for X86 (and recently PPC support is starting to emerge) the LinuxBIOS projects is an attractive alternative especially for larger-footprint X86 based systems (see the section on linuxbios)

LILO

The probably most used boot-loader for Linux is LILO , the Linux loader. It is available for x86 and also for 6xx power-pc. This boot-loader is extensively configurable and has a few featurs that are of great interest for embedded systems. A general list of LILO featurs:

- Boot from more or less any block-device (HD,floppy,DOC etc.)
- Configuration of the graphics mode
- Provide a graphicall boot-menu with boot-image selection
- Boot prompt for passing kernel arguments
- Can be protected in a limited maner by password

Especially for embedded systems the ability to boot an image once only and then fall back to the previous setup `lilo -R IMAGE_NAME` is atractive as it allows to test a new image in the field and in case of failure the local personell must do no more than cycle power. Other recent development like the graphics boot-screen are marketing featurs, but technically not that important, it does allow allow to be a bit more verbose giving the image selection more meaningfull strings like `Linux Kernel 2.4.16` instead of only `linux` or some cryptic string like (rtl32).

LILO also has a build in diagnostics if the boot-loader itselfe fails presenting only L, LI, LIL in case that one of the steps in the boot-strap process (loading of primary boot-loader (L), executing primary boot-loader (LI), loadingsecondary boot-loader (LIL), executing secondary boot-loader (LILO:). This allows to diagnose quite precisely where the system is failing even within the boot-loader start up.

LILO is available on more or less any Linux distribution you can get.

GRUB

The GRUB (GRand Unified Bootloader) boot-loader, originally developed by Erich Stefan Boleyn, and is not a GNU project (if your acronym starts with G then the chances are your project will end up as a GNU project...). Currently only x86 platforms are supported and there don't seem to be plans for ports (let me know if there are !).

Main featurs of GRUB:

- Multiboot setup (including non-multiboot OS's)
- Initialization of RAM and alows access to any storage media (no geometry dependance like in lilo)

- Human redable config scripts (the definition of human redable varies greatly though)
- Menu and commandline interface
- Support for network boot and network download of images

The GRUB bootloader is actively being developed and is available on many of the common distributions or you can get it from `ftp://ftp.gnu.org` .

PPCBoot

PPCBoot is a boot-loader for embedded power-pc boards. It is an ongoing effort and covers a relatively large number of boards by now (more than 80). As power-pc based systems don't have a BIOS like x86 systems a boot-loader must do much more low-level work. Basically this could be done in the Linux kernel system initialization files that are prepended to the compressed kernel, but this would not be very user-frindly .The main featurs of PPCBoot are:

- verbosity of the pre-kernel boot process
- Basic hardware initialization (memory/flash,ethernet and serial port)
- allows for writing to flash and loading via network
- provides a boot prompt to pass kernel commandline agruments, including variable expansion and access of config data
- Allows to store boot-parameters in non-volatile media (DOC/NVRAM)
- Offers a b̈oot-shellẅhich allows extensive configuration of the boot-setup at the ppcboot-prompt.

PPCBoot is maintained by Wolfgang Denx and is available at `http://ppcboot.sourceforge.ne`

U-Boot

U-Boot (Version 0.4) - the next 'unified' bootloader - this time for embedded systems. Stemming from PPCBoot and (TODO: what ARM-boot-loader went into u-boot ==) it has gained fairly large acceptance and is replacing PPCBoot (advocated by the former PPCBoot developers) and other embedded boot-loaders.

U-Boot supports fat,msdos and jffs2.

jffs2 support in u-Boot is a read only (re)implementation of the file system from Linux with the same name. U-Boot provides the folowing basic commands to access jffs2 boot partitions from the U-Boot cli.

- fsload - load binary file from a file system image
- fsinfo - print information about file systems
- ls - list files in a directory

nand-flash devices are well supported with extensive read/write/inspect commands.

Suppored hardware:

- i386 - (limited to ELAN SC520 at time of writing)
- Motorola - mpc5xx,mpc8xx,mpc824x,mpc8260,mpc74xx
- IBM - ppc440,ppc405
- Intel - pxa, sa1100, arm720t, arm920t, at91rm9200
- mips (seems very limited though)

U-boot is to be expected to gain even larger acceptance in the neer future and become a 'standard' boot-loader for small-footprint embedded GNU/Linux systems.

Syslinux

Storage media on embedded systems often come with a msdos filesystem on the media (e.g. DOC,CF), so for systems that want to use this msdos filesystem directly for booting a Linux bootloader called syslinux is available that will allow booting from FAT12 partitions. Syslinux has some nice features that come along with it:

- easy to configure for multi-boot systems via syslinux.cfg (ascii text file)
- allows assigning the function keys to text-files to present additional boot-time information for the user.
- simple to install with the syslinux command from Linux (the actual boot-loader is ldlinux.sys a DOS program)
- allows passing a kernel command-line at the boot prompt
- allows a specified boot-delay (time-out in syslinux.cfg) for auto-boot.

Syslinux was originally used quite heavily for Linux install/boot-floppies. It is still supported and is available on almost all common Linux distributions. With the time-out set to 0 the system will boot immediately, not allowing the user

to pass any parameters (but in this setup you also have no access to the help-screens). A disadvantage of syslinux is that the default boot-image is statically set in syslinux.cfg on the FAT12 boot-media, which means that if a boot fails you need qualified intervention (a simply power-cycling will not do), and you need a local console with a keyboard !

A further disadvantage of syslinux is that the FAT12 boot-medium is quite limited with respect to file-names and permissions. These limitations need to be considered for systems with high security demands.

25.5.2 Networked Systems

Network capabilities was one of the early strengths of linux - and very early in the developement of Linux, specialized Linux distributions for discless clients have evolved. XTerminals based on low end commodity component computers have been around quite a while, from which sepcialized systems like the Linux Kiosk system evolved as an example of embedded Linux running via NFS-root filesystem. In its latest version the Linux kernel is fully adapted to boot over the network and run via nfs-root filesystem, allowing for inexpensive and easy to configure embedded systems ranging from the noted kiosk system to embedded control applications that will boot via network and then run in a RAMDISK autonomously. The ability to operate in a discless mode is not only relevant for the administration, but also important for operation in harsh environments on the factory floor where harddiscs and fans are not reliable. A further usage of the network capabilities of embedded Linux is allowing for a temporary increas of 'local' resources by accessing remote resources, may this be mounting an administative filesystem adding an nfs-swap partition (a cruel thing to do...) or simply using network facilities for off-site logging. The network resources of Linux allow moving many resources and processing tasks away from the embedded system, thus simplifying administration and reducing local resource demands.

Performance

Performance issues with nfs-root filesystems and nfs mounted filesystems will rarely be a critical problem for embedded systems, as such a setup is never suitable for a mission-critical system or a system with high-security demands. Nfs-server and client in the Linux kernel is very tolerant towards even quite long network interruptions (even a few minutes of complete disconnection normally will be managed correctly), but this tolerance does not eliminate the performance problems and nfs-root definetly is only suitable for systems where the data-volume transfered is low. A special case might be using nfs-root filesystems for development purposes, this is a common choice, as it eliminates resource contraints related to storage media and simplifies development. Development on nfs-root filesystems, though, must exclude benchmarking and reliability tests

as the results definitely will be wrong. A stable nfs-root environment can offer a filesystem bandwidth well above a flash-media. On the other hand heavy nfs-traffic on an instable network or a highly loaded network will show false-negative results.

Secutirty of NFS

The nfs-filesystem does not have the reputation of providing a high level of security. So nfs-root systems should not be used in areas where network security is low, or on critical systems altogether (for a Kiosk system it may be well suited though). There are secure solutions for network file-systems, like tunneling nfs or SMB via a VPN, but these do not allow for booting the system in this secure mode (at least not to my knowledge). Also SMB, which is a state-full protocol is clearly better than nfs, but again, I don't know of any bootable setup providing something like smb-root. For systems that might use a local boot-media and then mount applications, or log-partitions over the network both SMB and tunneled NFS are possible with an embedded GNU/Linux system. A further possibility that may be feasible for some setups is to use advanced network file-systems like CODA that allow better access-control all the way to encrypting transfers.

25.5.3 RAMDISC Systems

RAMDISC systems are not Linux specific, but the implementation under Linux is quite flexible and for many embedded systems that have very slow ROM or media with a relatively low permissible number of read/write-cycles, a RAMDISC system can be an interesting solution. RAMDISCs reside in buffer cache, that is, they only will allocate the ammount of memory that is currently really in use. The only limitation is that the maximum capacity is defined at kernel/module compile time. The RAMDISC itself behaves like a regular block-device; it can be formatted for any of the Linux filesystems and populated like any other block oriented storage device. The specialities of Linux are related rather to the handling of the buffer chache, which is a very efficiently managed resource in the Linux kernel. Buffers are allocated on demand and freed only when the amount of free memory in the system drops below a defined level - this way the RAMDISC based filesystem can operate very efficiently in respect to actually allocated RAM. To operate a RAMDISC system efficiently an appropriate filesystem must be chosen - there is no point in setting up a RAM-disc and then using reiserfs (atleast in most cases this will not be sensible) a slim filesystem like minixfs, allthough old will be quite suitable for such a setup and yeald and efficient use of resources (imposing minor restrictions with respect to maximum filename length and directory depth).

Performance

One of the reasons for using a RAMDISC is file-access performance; a RAMDISC can reach a read/write bandwidth comparable to a high-end SCSI device. This can substantially increase overall system performance. On the other hand, a RAMDISC does consume valuable system-RAM, generally a quite limited resource, so minimizing the filesystem size at runtime in a RAMDISC based system is performance critical. It is a slight exaggeration, but doubling available system-RAM in a low memory setup can improve overall performance as much as doubling CPU speed!

A nice feature available for Linux is to not only copy compressed filesystem images to a RAMDISC at boot time, but to actually let the kernel initialize a filesystem from scratch at boot-up and populate it from standard tar.gz archives thereafter. The advantage of this is that the boot-media can contain each type of service in a separate archive, which then allows safe exchange of this package without influencing the base system. Naturally, exchanging the base archive or the kernel is still a risk but at least updating services - which is the more common problem - is possible at close to no risk. If such an update fails, you just login again and correct the setup. With a filesystem image you generally have to replace the entire image; if this fails, the system will not come back online, and a service technician needs to be sent on site to correct the problem. To put the additional RAM requirement into relation to the services - a system providing a Linux kernel and running SSHD, inetd, syslogd/klogd, cron, thttpd, and a few getty processes will run in a 2.4MB RAM-disc, and require a total of no more than 4MB RAM (2.2.X kernel based on glibc-2.0.7).

Resource optimization

When using a RAMDISK system, a few optimization strategies are available that are hard to use in general systems or desk-top systems. These optimizations are related to the files in a RAMDISC system only have a 'life-span' limited to the uptime of the system; at system reboot the filesystem is created from scratch. This allows removing many files after system boot up: init-script, some libs that might only be required during system startup and kernel modules that will not be unloaded during operation after system initialization has completed. The potential reduction of the filesystem is 30-40% on test-system built (e.g. MiniRTL).

A sometimes noted disadvantage of the RAMDISK implementation is that its upper bounds is statically set, other RAM based filesystems (e.g. ramfs, tmpfs) dynamically adjust to the size requested. Although this can be useful in some situations one must make sure that a user-space caused file-system flood (dd if=/dev/zero of=/tmp/garbage...) will eat up all available RAM and the system would hang.

A newer implementation of a RAM-residing filesystem is tmpfs (also some-

times still referred to as shm fs), tmpfs grows and shrinks with the files stored and can swap unneeded pages out to swap space. In a limited manner the above pitfall holds true for tmpfs as well - as the size is not statically fixed a incorrect size option passed at mount time can cause problems. Also one should be aware of the fact that the permissions of the tmpfs mount point are settable with module parameters and thus need to be taken care of by the initialization scripts. This just means that tmpfs is more flexible but requires additional attention when used to make its operation safe.

As neither ramfs nor tmpfs can be used for root-file-systems (at least at time of writing I know of no procedure comparable to creating a root-filesystem at boot-time in a RAMDISK) they can only be used in addition to some bootable file-system on an embedded device.

Security

As everything else, the choice of the system setup also has security implications, a few of these with respect to RAMDISC systems should be noted here. System security and long term analysis relies on continuous system logs, writes to RAMDISCs are quick, but to an off-site storage media or a slow solid-state disc are delayed, system logs may thus be lost. A possible work around is to carefully select critical and non-critical logs, writing these along with other critical status data to a non-volatile media (e.g. NVRAM). This solution is quite limited as, in general, no large NVRAMs will be available. Alternatively, logfiles may be moved off-site to ensure a proper system trace, as access may not be possible after a system failure. When writing logs to a non-volatile media like a flash-card locally one needs to consider the read/write cycle limitations of these devices, as letting syslogd/klogd write at full-speed to a logfile on such a media can render it useless within a few months of operations, making in hardly better than off-site logging.

A clear advantage of RAMDISC based systems is that although the filesystem modifications are volatile as - is the entire system - a 'hack' would be eliminated by the next reboot, giving a safe although invasive possibility to relatively quickly put the system into a sane-state of operations. To enhance this feature, access to the boot-media can be prevented by removing the appropriate kernel module from the kernel and deleting it on the filesystem. In case the boot-media needs to be accessed for updates, the required filesyste/media kernel-modules simply can be uploaded to the target and inserted into the kernel. This strategy makes it very hard for an unauthorized user to access the systems boot-media unnoticed. A reboot puts the system in a sane-state, as noted above - a system can also be configured to boot into a maintenance mode over the network, allowing for an update of the system. These methods are quite easy to implement. For example, such a dual-boot setup RAM-disc or Network, requires no more than a second kernel on the boot-media (¡= 400K) and a boot-selection that is configurable (syslinux, grub, lilo etc.) on the

system. RAMDISC based systems can be a security enhancement, if setup is done carefully.

25.5.4 Flash and Harddisk

Embedded systems need not always be specialized hardware - even if many people will not recognize an old i386 in a midi-tower as being an embedded controller - this can be a very attractive solution for small numbers of systems, development platforms and for inexpensive non-mobile devices. The processing power of a 386 at 16 MHz is not very satisfactory for interactive work, but more than enough for a simple control tasks or machine monitoring system. The ability to utilize the vast amount of commodity components for personal computers in embedded systems is not unique to embedded GNU/Linux, but Linux systems definitely have the most complete support for such systems, aside from being simple to install and maintain.

Harddisk based systems

Obviously the last mentioned method is only acceptable for systems that don't have low power requirements and can tolerate rotating devices, that is, are not to operate under too rough conditions. In these cases, the advantage of linux supporting commodity PC components may be a relevant cost-factor, as especially for prototype devices and those built in very low numbers, these components simplify system integration substantially (no special drivers, no non-standards system setups required). Aside from these spcialized systems, harddisc based systems are also interesting for development platforms, as they elliminate the storage constraints that are imposed on most embedded systems. And, with there ability to use swap-partitions on such a setup offer an almost arbitrary amount of virtual-RAM (although slow) for test and developement purposes.

Flash/solid-state discs

Solid state 'discs' have already been available for Linux in the 2.2.X kernel series. Obviously the IDE compatible flash-discs were no problem; other variants like (CFI-compatible,JEDEC or device spceific Flash-devices) were more of a problem, but the MTD project now has incorporated these devices into the Linux kernel with the 2.4.X series in production quality. The restrictions for some of these media do stay in place, that is, that they have a limited number read/write cycles available (typically in the range of 1 to 5 million write cycles - depending on the technology used and environment conditions as well as operational parameters). This can be a problem if systems are not correctly designed. A file system and the underlying storage-media tend to erase/write some areas more often than others (eg. data and log files will be written more

often than applications or configuration files, naturally the load can be very high in all temporary storage areas so the storage media may wear out faster depending on the systems layout. wear leveling strategies have been design to reduce this "hot-spot burnout" but this generally means data around to level out the wearing and thus reducing read/write performance of the media.

Imagine a swap-partition on flash or the system log-files with syslogs parameters not adopted; such a flash device could run into problems within as little as three months! When using a solid-state media with limited read/write cycles, filesystem activity should be reduced, write logfiles at long intervals, write data to disc in large blocks, make sure temporary files are not created and deleted at high frequency by applications. Taking the read/write limit into account, the effective life-span of such a system easily can be extended to years. If high frequency writes are an absolute must, then the usage of RAMDISCs for these purposes is preferable.

Since solid-state based systems generally don't loose their data at reboot, one must also take care of data accumulated in temporary files and especially in logfiles. For this purpose some sort of cron-daemon will be required on such a system, allowing for periodic cleanup. Also, in general, a non-volatile root-filesystem will be 30-40% larger than a volatile RAMDISC based system - if file integrity checks are necessary (as a reboot will not put the system back into a sane-state after file-corruption or a attack on the system) the filesystem can be double as compared to a RAMDISC based system.

Alternatives to delayed read/writes to devices with limited read-write cycles after to use filesystems that implement wear leveling like jffs and jffs2 (or use devices that implement wear leveling in hardware like DOC or some PCMCIA cards). Generally this should be taken into account for any devices that don't implement wear leveling on the hardware level (like Compact Flash and Smart Media...correct me if I'm wrong on this...). An no - journaling filesystems don't automatically guarantee wear leveling. They will protect the filesystem against power-fail situations which older filesystems like minix or ext2 don't handle very well - especially if the failures occurs during write cycles - but journaling filesystems will also show hot-spots with respect to read/write cycles that can reduce the life span of some devices.

One characteristic of solid-state devices that must be taken into account is that they are relatively slow (although faster devices are popping up lately). This has implications on the overall system performance as well as on the data-security of items written to disc. Solid-state discs will often exhibit a data-loss on the items being processed at the time of power-loss, though this does not though influence the integrity and stability of the filesystem itself. So in a solid-state disc based system, critical data will have to be written to a fast media if it is to be preserved during a power-loss.

The generally low performance of solid state discs with respect to read/write bandwidth can be overcome in some setups by having a "swap-disk" located in RAM. This might seem surprising that reducing system RAM and putting

some into a swap-partition can improve performance, but this is the case due to the different strategies that Linux uses to optimize memory usage - swapping to a slow media would hurt performance greatly - swapping to a fast media will improve swap performance and at the same time the Linux kernel will modify its optimization strategy to use the reduced RAM as good as possible. The implementation of such RAM-swap-DISKS can be done with current MTD drivers using slram on top of mtdblock. Slram provide access to the memory area reserved (by passing a mem= argument to the kernel limiting the kernels memory to less than physically available), mtdblock provide the block devices interface so that this memory area can then be formated as swap partition on system boot.

Flash technologies

In this section a brief introduction to flash-technology is given , this is necessary to understand the difference in system setups between standard desk-top systems and flash based systems. This has clear indications for the selection of filesystems and boot/operational concepts. As this knowledge is probably not that wide spread, this slightly off-topic section is inserted.

Flash

As embedded system designers don't like rotating media, solid state devices have been becoming very common - they provide high storage density at low power consumption and relatively low expenses. The two major types of Flash are the directly accessible NOR flash, and the newer, cheaper NAND flash, addressable only through a single 8-bit bus (for both data and addresses) and additional control lines.

Unlike RAM chips flash chips are not able to simply set bits to 0 or 1, each bit in an erased flash is set to a logical one, write operations set it to logic 0.Due to this operating a flash device requires a separate process to take care of this erasing / s̈etting to logic 1; this process is done by the cleaner or garbage-collection.

Flash chips are arranged in blocks 128KB (NOR) and 8KB (NAND), this is the reason for the difference in reserved area in jffs2 for the garbage-collection, which is currently five such erase blocks - work is ongoing to reduce this in future versions of jffs2. Resetting bits from zero to one cannot be done individually, but only by resetting (or "erasing") a complete block. The lifetime of a flash chip is measured in such erase cycles, with the typical lifetime being about 100,000 to 1,000,000 erase operations. To ensure that no one erase block reaches this limit before the rest of the chip, most users of flash chips attempt to ensure that erase cycles are evenly distributed around the flash; a process known as wear leveling. As this wear leveling requires to move data around on the device ,which is taken care by the garbage-collection thread, and access to flash devices is relatively slow, overall throughput of such devices is low compared to hard-disks. Typical

values are in the range of 200 KB/sec to 800 KB/sec, claims of 20MB/s in burst mode can be found in the Internet, no idea how these values are measured... and for persistent write operations the given values of ¡800KB/s seem to be reasonable (corrections appreciated).

A further difference between NOR and NAND chips is that the later is further divided into "pages" (typically 512 bytes), each of which has an extra 16 bytes of "out of band" storage space, intended to be used for meta-data or error correction codes. In recent MTD releases this is available by selecting CONFIG_MTD_NAND_ECC (software-based ECC). It can detect and correct 1 bit errors per 256 byte blocks. The NAND flash is written by loading the required data into an internal buffer one byte at a time, then issuing a write command. While NOR flash allows bits to be cleared individually until there are none left to be cleared, NAND flash allows only ten such write cycles to each page before leakage causes the contents to become undefined until the next erase of the block in which the page resides. The number of writes befor requireing a r̈efreshb̈ erasewrite cycle can be as low as 1 write cycle to the main data area, and 2 write cycles to the spare data area on some NAND devices.

Flash Translation Layers

Due to UNIX filesystems expecting block devices and flash not being organized in block like a hard-disk, a brute-force emulation approach has been common. That is, one introduced an additional layer between flash-device and the filesystem (e.g. ext2) that would emulate a normal block device with standard 512-byte sectors.

The simplest method of achieving this is to use a simple 1:1 mapping from the emulated block device to the flash chip, and to simulate the smaller sector size for write requests by reading the whole erase block, modifying the appropriate part of the buffer, erasing and rewriting the entire block, which one easily can imagine not to be an extremely efficient way of doing things. This approach provides no wear leveling is extremely unsafe because of the potential for power loss between the erase and subsequent rewrite of the data, and reduces the bandwidth of flash devices noticeably. However, it is acceptable for use during development of a file system which is intended for read-only operation in production models. The mtdblock Linux driver provides this functionality, slightly optimized to prevent excessive erase cycles by gathering writes to a single erase block and only performing the erase/modify/write-back procedure when a write to a different erase block is requested.

To emulate a block device in a fashion suitable for use with a writable file system, a more sophisticated approach is required. To provide wear leveling and reliable operation, sectors of the emulated block device are stored in varying locations on the physical medium, and a "Translation Layer" is used to keep track of the current location of each sector in the emulated block device. This translation layer is effectively a form of journaling file system.

The "Flash Translation Layer" [FTL] (which is part of the PCMCIA stan-

dard). More recently, a variant designed for use with NAND flash chips has been in widespread use in the popular DiskOnChip devices produced by M-Systems. Unfortunately, both FTL and the newer NFTL are encumbered by patents – not only in the United States but also, unusually, in much of Europe and Australia. M-Systems have granted a license for FTL to be used on all PCMCIA devices, and allow NFTL to be used only on DiskOnChip devices.

Linux supports both of these translation layers, but their use is deprecated and intended for backwards compatibility only. FTL/NFTL are also not very efficient - it inserts an additional layer between the physical device and the filesystem, directly talking to the device simply is more efficient, and with jffs/jffs2 being available the problem of wear leveling AND journaling is resolved cleanly.

Combined systems

If boot up time is critical then a large romfs (uncompressed) or cramfs (compressed but indexed - so no processing on mount required) and a relatively small jffs2 will make it boot faster, as jffs2 does not need to scan the entire device. One must concider though that wear leveling is limited to the jffs2 partition in this setup. Other combinations may be to have msdos filesystems to boot from (if your broken BIOS only accepts an msdos-fs for booting..) and then have a real filesystem on a second partition via flash-translation-layer or directly using jffs/jffs2.

Other posible combinations are to have the kernel and a compressed initial ramdisk (initrd) on the raw media and a s̈lowf̈ilesystem on a part of the flash device.

Optimizations of this type make sense if the devices in question are extreemly small (¡4MB) or if boot-time is critical.

A second reson for combining devices is that NAND-Flash is cheaper but you can't easaly boot from NAND directly. There are two ways to solve this:

- Use a CPLD as a minimum F̈lash-controller; which provides you access to the first NAND-Page, which must then contain a small bootstrap code. This is a cood solution if you want to be part of the CPLD ḧype...

- Use a small and cheap 1MB FLASH for bootload which can also contain a compressed kernel Image. Start from there and mount a filesystem on the NAND-Flash. This might not sound very creative, but it's a solid solution using MTD drivers for both devices.

Naturally other combinations may be sensible as well - the main issue here is that combining devices can optimize costs, they will though lead to more complex systems. Solutions that require technology that is not widely deployed and thus well tested will increas testing and maintenance, so probably the second example is the better and safer way to go.

25.5.5 Linux in the BIOS for X86

This section is hard to place - the projects described here merge the boot-operation and the Linux kernel operation together so they are placed here at the end of the operational concepts.

Many embedded X86 based devices require a specific BIOS, this can be quite an expensive part of an embedded project as it not only requires substantial initial investment (if you let a BIOS provider role your custom BIOS) but it also includes per device royalties. Last you need to add a storage device (EPROM or the like) to the system that is only used for the boot process and after that is totally useless as Linux does not use BIOS-calls in the kernel at all (only setup.S/video.S and bootsect.S have BIOS calls in them which is not part of the kernel but are a minimum "boot-loader" prepended to the compressed-kernel). The Linux BIOS project allows you to boot Linux directly which improves boot-times dramatically. If you want to give the bzimage copy orgy of an X86 boot a close look, check out Alesandro Rubinies Linux Device Drivers.

Currently there are two projects around to boot Linux directly from the cold-box, the LinuxBIOS project and ROLO. LinuxBIOS is well under way to gain support for a relevant number of motherboards for X86 (UP and SMP systems) aswell as recently announced work on PPC systems may expand this interesting project to new architectures.

As the BIOS is a proprietary and generally roaylty based code part of a systemit may well be a cost issue to concider the LinuxBIOS project.

ROLO

As ROLO is not using any BIOS calls that provides a hardware abstraction, it naturally will be hardware specific - the good news is that the hardware specific part is quite small. The original implementation to be found on the Internet is based on AMD's SC520 CDP (eval-board for the SC520), embedded devices currently supported are:

- Syslogic NetIPC `http://www.syslogic.ch/`
- Intels i386EX eval board `who know the link ??`
- AMD Elan CDP `link ??`

LinuxBIOS

LinuxBIOS is an Open Source project aimed at replacing the normal BIOS with a little bit of hardware initialization and a compressed Linux kernel that can be booted from a cold start. The project was started as part of clustering research work in the Cluster Reseach Lab at the Advanced Computing Laboratory at Los Alamos National Laboratory. The primary motivation behind the project was the desire to have the operating system gain control of a cluster node from

power on. Other beneficial consequences of using LinuxBIOS include needing only two working motors to boot (cpu fan and power supply), fast boot times (current fastest is 3 seconds), and freedom from proprietary (buggy) BIOS code, to name a few. Having the BIOS code available in-house and it being based on known and open technology like Linux/RTLinux allows to respond to bugs and adopt to security demands at a much more fine grane level than would be posible with a proprietary BIOS (I remember long lists of BIOS-passwords floating around the internet..)

Suported main-boards:

- Intel L440GX+
- Winfast 6300
- Procomm BST1B -based mainboards
- Gigabit GA-6BXC
- SiS 730 (i.e. K7) chipset
- VIA VT5292A
- VIA VT5426
- ASUS CUA ALI TNT2 (Acer
- M1631/M1535d chipset)
- (TODO: update to lates list)

As you can see from the list of devices these are not exactly typical embedded systems (allthough a cluster in a lunch-box is to be concidered an embedded cluster - the list is though rapidly expanding and work on PowerPC support is also under-way (instable at time of writing)

25.6 Compatibility and Standards Issues

The term compatibility has been widly misused, OS's claiming to be 'compatible' as such - without stating to what they are compatible. So first a clarification as to how this term is being used here. Compatibility between embedded OS and desk-top development systems is one aspect here, this compatibility being on the hardware and on the software level, aswell as on the administrative level. Byond that level of compatibility there is also a conceptual compatibility, which is of importance not only for the development, but also to an even higher degree, for the evaluation of systems. The compatibility of embedded Linux to desk-top development systems as understood here, is defined as the ability to

move executables and concepts from the one to the other without requireing any changes. This does not mean that some changes might then be made for optimization resons but there is no principle demand for such changes. As an example one might consider a binary that executes on the desk-top and the embedded system unmodified, but in practice would be put on the embedded system in a stripped version - this is no conceptual change though.

25.6.1 POSIX I/II

The blessings of the POSIX standards have fallen on GNU/Linux - as much as these standards can be painful for programmers and system designers, they have the benefit of allow clean catagorizations of systems, and they describe a clear profile of what is required to programm and operate them. This is a major demand in industry, as evaluation of an OS is a complex and timer-consuming task, so POSIX I cleanly defining the programming paradigma and POSIX II (not so cleanly) defining the operator interface, simplify these first steps.

The RTLinux API is a POSIX PSE 51 based threads API, provides a subset of POSIX interface targeted specifically at minimum realtime systems. As the POSIX threads are widly in use, moving to RTLinux is simplified greatly. The PSE 51 standard complience not only simplifies the programing task but also allows to resort to a well established knowledge base during the design phaas. RTLinux provides the folowing sumary of POSIX functions to the programer:

- Time related functions
- Basic p_thread functions
- synchronisation primitives (mutex,semaphors)
- POSIX condition variables
- Non-portable POSIX extensions. These rtlinux specific extension, that simplify your life

If one is familiar with POSIX threads then it should be simple to move on to the real-time capabilities that RTLinux provides, this not only is a efficiency question but naturally a well sepcified and commonly used API improves security. This improvment is due to the potential pitfalls of POSIX threads being well documented which increases the ability to evaluate the security implications of a programming descision.

RTAI has very limited POSIX pthreads complience, notably some pthread_ functions folow POSIX pthread syntax but not there semantics, at this point RTAI seems not to be anticipating POSIX complience (although the debate is on the agenda all the time...).

Basically it must be stated that the POSIX pthreads API (including that of the PSE 51 minimum realtime profile) does show some clear defizites for

embedded realtime systems, notably lacking the notion of periodic threads, complexity of signals (ugly to use) and limitation as to optimizing close to the hardware (cpu-affinity, per-thread fpu-managment etc.)

Still the POSIX pthreads API clearly seems to be the best available standard API for embedded realtime systems.

- well documented
- sufficient trainings material (tutorials etc.) available
- wide spread in industrial applications
- well investigated concepts and software design patters available

25.6.2 Network Standards

Aside from the important POSIX standards, GNU/Linux also follows many other standards, notably in the network area, where all major protocols are supported. Supported standards include the hardware standard for Ethernet Token-Ring FDDI, ATM etc., and the protocol layers TCP/IP, UDP/IP, ICMP, IGMP, RAW etc.. This standardization level allows a good judjment of a embedded Linux system at a very early project stage, and at a later stage simplifies system testing a lot.

25.6.3 Compatibility Issues

The demand for compatibility of embedded systems and desk-top development systems touch far more than only the development portion of an embedded system. As much of the operational cost of systems lies in the administration, and a major issue, evolving even strongly now, is system security - the question of compatibility is very high ranked. The more systems become remotely accessible for operation administration, even for a full system update over the internet, the more it becomes important to have a well-known environment to operate on. This is best achieved if the remote system behaves "as expected" from the standpoint of a desk-top system for which developers and administrators have feeling for - even if many people in industry will not like this 'non-objective' criteria, it is an essential part. And looking at a modern photo-copy machine one will quickly have the impression that this is a miniaturized XTerminal that one is looking at triggering expectations on the side of the user.

Development related

During the development process for an embedded system there are a few distinct states one can mark:

- system design - one of the hardes steps in many cases.

- specification system security policy
- kernel adaptation (if necessary somtimes simply a recompile and test)
- core system development - a root-filesystem and base services.
- custom application development and testing

The first step is the hardest for a beginner, and having a desk-top Linux system to 'play' with can enormously reduce this effort. It is very instructive to set up a root-filesystem and perform a change-root (chroot) to that directory, gathering hands-on experience for the system, systematically reducing executables, scripts, libs, etc. A highly compatible system obviously is a great advantage here. Directly related to the first step, a threat analysis and the security specification should folow, my personal experience with industir and telekom projects up to now has been that this issue was neglected if not completly ignored ! It is a very expenive and time-consuming task to add security requirements after a system was completed. As an example of why this may become so expensive concider the extensive CPU demands for a resonably secure encryption of network packets - many embedded systems don't provide enought extra CPU power to allow adding this later, mandating to upgrade the syatems ahrdware due to security demands, from this it is obvious that a late design of security issues is a clear project managment error. As standard encryption methods are well documented and benchmarked the resource demands for the security related design steps can generally be well estimated.

The kernel adaptation phase can be simplified, if a desk-top system with the same hardware architecture is available (especially for x86 based systems this generally is the case), allowing compiling and pre-testing the kernle for your hardware. The third step - actually building the root-filesystem, is not as simple as it might sound from the first step described above. A root-filesystem needs to initialize the system correctly - a process that can not only be hard to figure out - but, also hard to debug, if the system has no direct means of talking to you (it can take a month until the first message appears on the serial console of some devices...). Designing a root-filesystem requires that you gain understanding of the core boot-process. To gain this understanding a desk-top system is hardly suitable, resorting to a floppy distribution (linux-router-project or MiniRTL) can be very helpful.

Where compatibility between your desk-top and the target system can save the most time - is when your application runs on your desk-top, if the debugging and first testing can be done on a native-platform. The biggest problems are encountered during developement with cross-compiler handling and cross-debugging on targets that don't permit native debugging. Even though there are quite sofisticated tools available for this last step, a native platform to develop your application is by far the fastest and most efficient solution (although not allways possible).

Operation Issues

Hardware and development expenses are a major portion for the producing side of a system. For people operating embedded systems, maintenance and operational costs are the major concern in many cases. Having an embedded system that is compatible to a GNU/Linux desk-top system simplifies not only administration and error diagnostics, but can substantially reduce training expenses for operational personnel. Compatibility is also relevant for many security areas. It is hard to implement a security policy for a system with which operators have little hands-on experience. At the same time there are few documents to reference on such a security policy for proprietary systems. Being able to apply knowledge available for servers and desk-top improves the situation and opens large resources on the security subject for operators of embedded systems. One further point that can be crucial is the ability to integrate the system into an existing network infrastructure. The immense flexibility of embedded linux in this respect simplifies this task a lot.

25.6.4 Software Lifecycle

TODO: embedded realtime specifics of V-model vs. throwaway prototyping

25.7 Engeneering Requirements

Embedded GNU/Linux is not quite as simple as there proprietary counter parts, in most proprietary embedded systems one need not select between many different implementations for a given problem statement, in GNU/Linux it is not uncommen to find a dozen projects that will do (i.e. embedded web-servers: thttpd,boa,minihttp,sh-httpd,etc) - this mandates that a project managment engeneer in the are of embedded GNU/Linux have a resonably well established know-how on the issues of tool-chains, especially those tools that allow safe platform independant source development. As a starting point of these engeneering capabilities we would see:

- source managment (cvs,bk)
- GPL project understanding
  - how to join into GPL projects
  - what can be contributed- what can't be contributed
  - how much time should engeneers comit to community related issues (mailng listsetc.)
- development environmetn issues
  - desk-top development distribution selection (SuSE,RH,etc)

  - work environment selection/interface tools (i.e. all use the same interface tools, like xgdb OR ddd - mixing is expensiv)
  -

- tool-chain
  - sed,m4,gawk,perl,sh
  - automake,autoconf,libtools,make
  - binutils,gcc,gcov,bgcc
  - gdb,kgdb,strace
  - benchmark tools
  - doc-tools (groff/nroff,tex,doc-book)
  - information presentation (web/ftp structuring and search facilities - this is also the engeneers part of web-presentations not the web-designers job!)
- maintenance strategy
  - design of softwaware update plan
  - release of part/full software to the open-source community - delegating maintenance to the user-group and reducing in-house efforts
  - managment and integration of improvments/patches to released technology/code
  - training/information of in-house personel on evolving technologies in the area of the project (embedded GNU/Linux is not only a fast developing technology-pool it also is very heterogenous)

This list is long - and for proprietary OS/RTOS it may sometimes seem shorter, our belive is that it is never substantially shorter, maby with the exception of the open-source/GPL related issues, it just is not made explicid. What should be emphasised is that there is a lot BEFOR gcc and quite a lot AFTER gcc, as very often job-profiles will list the core tool-set and maby some architectural requirements but neglect the scope of development related technologies. At the same time the list should make clear that no individual will fully provide all these experiences, so one needs to ecalculate a fair amount of training and 'unproductive' work-time for engeneers that move into embedded GNU/Linux. To give this some numbers, our experience, which is limited to a non-representative set of engeneers shows that entering the full scope of embedded GNU/Linux requires a time frame of 3-4 month if supported by training blocks for some of the critical technological topics (OS/RTOS kernel basics, tool-chain introduction, GNU project managment) aside from typically to be expected know-how (if you have to explain what priority inversion or a spinn-lock is 3 month for a RTOS developer could be tight...)

The full power and potential of embedded GNU/Linux can't be unleashed if there is no established tool-chain and OS-core know-how available in a project team and if the advantages of the open-source community are nut utilized.

25.8 Conclusion

In this section the strength and capabilities of embedded and distributed embedded GNU/Linux and real-time Linux systems have been scanned. My very personal belive is that we will see a move from dedicated standalone devices towards distributed embedded systems in the near future. GNU/Linux is able to provide the resources required for this challenging path, it gives the developers the tools to unleash there creativity.

The intention of this introduction to embedded linux resources was to allow judgement of the quality and limitations fo comercially offered dev-kits. From the complexity of available resources and the nature of independant open source projects developement kits are limited in many ways:

- limited in scope
- dev-kits must be general enough to satisfy many platrfoms which leaves little room for optimization
- build procedures are not standardized leading to complex integration of any components the vendor does not include
- often packages are modified to fit into dev-kits which breaks available patches and limits support by the comunity
- dev-kit bind unrelated packages to each other limiting the ability to utilze recent developments in the open source comunity
- dev-kits are a relevant cost factor
- dev-kit support if very limited as the complexity of the available resources does not allow vendors to realy support all packages
- modifications and limited communication between users of dev-kits, limit the bug-fix and testing capabilities
- update cycles can become very expensive, especially if update of a single package leads to the entire system having to be updated (limited compatibility of dev-kits with open source projects)
- dev-kits are generally build for ‘generic‘hardware configurations like i386, this limits the ability to utilize platform specific resources

- integration of dev-kits into company source managment structures can be problematic for legal reasons aswell as for handling reasons (its simply not posible to integrate dev-kits into arbitrary structures - atleas for most dev-kits this is not easy)

Although there are eal open-source'development kits like the ??LDK there are clear advantages of having the technological basis in house:

- freedom of selecting from the rich variety of available implementations, for almost any problem the open source community has developed multiple approaches.
- flexible to respond to updates/patches to specific packages
- flexibility of integrating non-mainstream patches in packages
- allow specific optimization
- allow for cooperate identity being represented in the embedded product

This set of arguments pertains to the dedvelopment kit itselfe, and could be resolved by contracting a company to build a custom specific dev-kit - but that is only a small part of the actual problem. The main isue is the handling issues of dev-kits:

- system level debuging requires knowledge of the installed packages and there interaction
- system evaluation and certification can't be done with a 'black-box'dev-kit, a real open source dev-it that provides the entire build process could do - but still requires the engeneering effort of understanding the build process
- errors resulting from an individual package missbehaving (i.e. IPsec or inetd) require access to the know how of package buildup to debug and correct it.
- compile time options influence the overall system performance
- developing the know how of package managment and integration at the system build step provides the basis for testing evaluation and debuging, developeing this know how along with the system design simplifies these tasks considerably
- system design issues are in the hands of the engeneers, this allows to adjust the system in an early design phase, which is essential for designing security policies and evaluation proceeures.

And last - but not least - the issue of project managment is touched as building your up a development envoironment is essential for a project - this development environment will typically include

- project specification
- security policy/specs
- too-chain
- debug facilities
- hardware related code
- hardware independant code
- documentation
- testing and validation/certification proceedures and applications

Some of these items are covered by dev-kits - but typically not all, and definitly they will not be adjusted (and often not adjustable) to company policy (source managment system), butilding up this development environment in-house not only builds up valuable know-how but allows a high level of flexibility for projects. It may be an exageration for small projects, but generally the handling related delays in a project are atleast in the same order as the technologically relateed delays. This was the original momentum that triggert the development of dev-kits and distributions. For embedded Linux projects it shows that standardisation is not as easy as it is for desk-top and server systems, although there are initiatives that may well help in this area ??SB??LCPS, which limits the value of dev-kits very clearly.

25.8.1 Borad support packages

A note on board-support packages, the limitations noted above, especially the limitation of generic builds and the issue of tight hardware dependancies in the embedded GNU/Linux world, have lead to the development of board-support packages. This are basically dev-kits with the hardware related issues resolved (and in many cases iven more proprietary than regular dev-kits...), but they nither solve the issue of package inflexibility nore the important issue of project managment related topics - in this sense they are a very marginal improvment over dev-kits, but may well be relevant for an early project state, especially in the evaluation phase of hardware selection .

25.8.2 summary

As a summary one can summarize the draw-back of dev-kits as breaking any top-down desing as it requires to 'specify within the bounds of existing dev-kit implementations'.

This may give the impression that dev-kits and board-support packages are concidered useless by use - this is not the case. We concider dev-kits usable for:

- early project state - hardware evaluation/selection
- throghaway prototyping
- reference platforms
- technology evaluation
- training of engeneers

and in the case of fully open source dev-kits/board-support packages, that is, distributions that provide the full build scripts from unmodified GNU sources (or include all relevant patches and documentation), are well suited as starting point for establishing the in-house know-how. The conclusion from the above though is clearly that developing a cooperate dev-kit with the in-house know-how basis seems like the most efficient approach to embedded GNU/Linux.

Appendix A

Terminology

Since there are many definitions floating around, a few key terms should be clarified. These are not to be considered 'authoritative', however they are the way they will be used in this document.

Hard Real-Time

A system will be considered "hard real-time" if it fulfills the following list of requirements. Missing any of these points means the system is not hard real-time.

- System time is a managed resource
- Guaranteed worst-case scheduling jitter
- Guaranteed maximum interrupt response time
- No realtime event is ever missed
- System response is load-independent

A system that can fulfill these criteria is deterministic with respect to the real-time applications running on it.

Soft Real-Time

In cases where missing an event is not critical, as in a video application where a missed frame or two is not fatal, a "soft real-time" system may do. Such a system is characterized by the following criteria:

- The system can guarantee worst-case average jitter
- Average interrupt response time will not exceed a maximum value
- Events may be missed occasionally

Soft real-time systems are statistically predictable, but a single event can not be predicted. Soft real-time systems are generally not suited for handling mission-critical events.

Non Real-Time

"Non-realtime" systems are the systems most often used. These systems are simpler and are able to utilize optimization strategies that are contradictory to realtime requirements, for example caching and buffering. Non real-time systems are characterized by:

- No guaranteed worst-case scheduling jitter at all
- No theoretical limit on interrupt response times
- There is no guarantee that an event will be handled
- System response is strongly load-dependent
- System timing is a unmanaged resource

Non real-time systems are unpredictable even at a statistical level. System reaction is highly dependent on system load. Non-realtime systems can use optimization strategies that are unsuited for hard or soft real-time systems.

Hard real-time systems will generally have slightly lower average performance than soft realtime systems, which in turn are generally not as efficient with resources as non-realtime systems. On the other hand, non-realtime systems are not at all predictable and soft-realtime systems are only statistically predictable. Only hard realtime systems are deterministic with respect to high-priority tasks. From the above definitions, it is clear that the border between non- and soft-realtime is difficult to define precisely. In general, these definitions will vary depending on the criteria that are emphasized when describing such a system.

Preemption

Halting a process in the middle of execution due to a higher priority process being ready to run is called preempting the process

Preemptive Kernel

If a process can be safely preempted during a system call then the kernel is considered preemptive.

Preemption Points

Code of the form:

```
if (higher-priority-task runnable ?) {
  invoke the scheduler;
}
```

in the kernel preemption patches (also referred to as low-latency patches) you can find this as:

```
  if ( current->need_resched ) schedule();
```

Priority Inversion

Blocking of a high priority process due to a low priority process, effectively lowering the priority of the high-priority process:

Priority inversion occurs when a low priority task locks a resource by acquiring a lock (mutex, semaphore, etc.) a medium priority task is runnable and a high priority task wants to aquire the lock that the low priority task holds - in this situation the low priority task will never get the chance to run and thus free the lock due to the medium priority task, which effectively means that the medium priority task is blocking the high priority task.

Priority Inheritance

If a process' priority is adjusted to that of a synchronization object that it locks to ensure that no priority inversion can occur.

Priority Ceiling

A PCP (Priority Ceiling Protocol) synchronisation object has a priority ceiling associated with it. No thread that has a priority higher than the ceiling, thus being "more important", can obtain the PCP synchronisation object. This ensures that a high priority thread can't blcok on a synchronisation object that is owned by a lower priority task. Deadlock prevention is ensured because all threads that were able to aquire the synchronisation object are granted a temporary priority equal to that set as ceiling of the synchronisation object.

System Call

A kernel function invoked via a library function to transfer control to the kernel so that a privileged operation can be performed on behalf of the calling process.

Critical Section

Any section of code that relies on the invariance of global objects (any non-local variables) for more than a single CPU instruction is said to be in a critical section.

Interrupt

Halting the CPU execution of the current process due to an electric signal that forces a context switch to handle the interrupt event.

Polling

Synchronous waiting for an event by probing in a loop until an event is found, typcially polling is used if very short intervalls between events is expected and thus the overhead of using interrupts is not desireable.

Reentrant Code

Code that can be preempted at any time, basically this means all global data objects in this code are accessed in an atomic way (either atomic - single cycle - CPU instructions, or by appropriately locking global object for exclusive access).

Atomic Instruction

Instruction consisting of only a single machine language instruction (i.e. `change_bit` on x86 evaluates to a single `btcl` assembler instruction), or a sequence of instructions protected by software synchronization primitives.

High Resolution Timers

This is not a very precisely defined term, ONE definition, and this is the one used in this document, is to call a timer that directly accesses the hardware timer resource (i.e. PIT, 8254, APIC-timer, etc) a high resolution timer, conversely a low resolution timer is a timer that is based on some hardware independent time base (i.e. jiffies) for reporting the time. This does NOT quantify the precision of the timer resolution per se, but generally high-resolution timers show resolutions in the order of micro-seconds to nano-seconds.

Time Stamp Resolution

On a realtime system it is generally quite irrelevant what the timer resolution is as this is generally much more fine grain than can actually be achieved on the process level, so the somewhat more relevant value is the time stamp resolution, which is the precision with which a point in time can be registered. As an example consider the 8254 timer chip on x86 platforms, its timer resolution is

838ns (normally it operates at 1.19MHz) but reading this chip is slow so two consecutive reads of the 8254 registers and the arithmetic required to calculate the time from the register values and handle overflow is about $10u$s on a i486, so on such a system the time stamp precision would be these $10u$s as this is the greatest precision with which an event can be time-stamped.

Spin-Lock

A synchronization primitive for concurrently executing processes where one process will wait in an active (running) state, for a resource. This means that this process will continuously poll the availability of the resource until it is available, this method is only efficient if used for resources that are held for a very short time (that is in the order of the context switch time for a given system) - it is also referred to as busy-waiting.

Fair Scheduling

On a general purpose operating system it is desirable that all processes including the lowest priority process are scheduled at some point - to ensure this not only the priority of a process but also its absolute run-time is taken into account granting more time to higher priority processes and less time to lower priority processes.

Fixed Priority Scheduling

In a fixed priority scheduling scheme a process has an invariant priority and will only execute if there is no higher priority task runnable - this can lead to unbounded delays of low-priority tasks. Fixed priority scheduling is the default policy for most real-time schedulers (i.e. default in RTLinux and RTAI).

Kernel Space

Execution context were a process has full access to the underlying hardware, all kernel space tasks in Linux are operating in the same address space.

User Space

Unprivileged context of execution in a private insulated memory area (virtual address space) that ensures that errors in memory access can't harm other user-space and kernel-space tasks.

RT Context

Execution context is under the control of the realtime executive.

Interrupt Latency

The time from assertion of a logical high signal on the CPU's interrupt line to the execution of the first instruction of the interrupt service routine.

Scheduling Jitter

The absolute time between when a task was scheduled to run and the point in time at which it actually started execution.

Lock Breaking

The insertion of lock-releas/lock-reaquire sequences in control paths that would otherwise have a resource locked for a very long time. By doing this the uninterruptible lcok times are reduced which improves the average system response in some cases.

Lock Granularity

Lock granularity describes how short in terms of execution time a protected or critical code section actually locks a synchronization object, and thus blocks other processes from executing. The most important lock in this respect is disabling and enabling interrupts. A system with a high lock granularity is a system that never block interrupts for a long period of time.

Over-committing Memory

If an OS assigns more memory to a single process than physical memory is actually available in the system it is said to have over-committed memory, this is a usable optimization for non-rt applications that will never actually use all of the allocated memory at the same time.

zero copy interface

allow information transfer by passing references only but without copying the actual data-items (memory locations), this is based on sharing memory between multiple processes via mmap'ing a physical memory location into the memory map of multiple processes.

Translation Lookaside Buffer (TLB)

A Translation Lookaside Buffer (TLB) is a table in dedicated memory that contains references mapping virtual to real addresses of recently referenced memory pages, it is used to speed up memory translation required in virtual memory systems like Linux.

just-in-time

Instead of simply starting individual processes irrespective of each others execution times, threads are implicitly associated with each other by setting the start-times of rt-tasks to the suspend times of the preceeding task (task-chaining) this is referred to as just in time scheduling as each time the scheduler is invoked there is ideally exactly one task that is runnable only.

Open Source

As this term is a key issue in this study we give a definition, even if this is not a very formal definition, it is what we mean in this document if we state something is open source.
Open source is a software that provides commented source code and accompanying concept documentation. It is not enough to dump 30MB of source to the printer to call a project open source. The issue is availability of technology, and technology is available if concepts are open, standards incorporated are open standards, resources required to utilize a technology are open to the public.

pthreads

The term pthreads refers to POSIX threads, more precisely, for threads API that anticipate full POSIX compliance. The usage of pthreads is common as the API generally has the form of pthread_. It should be noted that Linux threads available in the glibc package are often not referred to as pthreads as they are not strictly POSIX compliment. As the rt-threads implementations generally target POSIX compliance, even if not reached, the term pthread is used.

IRQ affinity

On multiprocessor systems interrupt management can be split among CPUs. To optimize the interrupt load distribution for rt-processes, interrupts can be assigned to a specific CPU for management.

Contention Protocol

A type of network protocol that allows nodes to contend for network access. That is, two or more nodes may try to send messages across the network simultaneously. The contention protocol defines what happens when this occurs. The most widely used contention protocol is CSMA/CD, used by Ethernet. Another example is CSMA/CA, used by CAN protocol.

Optimistic interrupt protection

"Optimistic interrupt protection" is a optimization of the fast-path - but not the worst case path in principal. The underlying assumption is that in most cases of critical sections, which are to be short, no hardware interrupt will disturb execution. This allows to optimize the system by not using the hardware interrupt masking capabilities on entry of the critical section but defers the masking of interrupts until a interrupt actually occurs, by introducing a software layer that checks if a given interrupt should be delivered or not.

Appendix B

List of Acronyms

PCP - Priority Ceiling Protocol
NFS - Network File System
RPC - Remote Procedure Call
TLB - Translation Lookaside Buffer
FPU - Floating Point Unit
MMU - Memory Management Unit
PIC - Programmable Interrupt Controller
APIC - Advance Programmable Interrupt Controller
PIT - Programmable Interrupt Timer
RTAI - Real Time Application Interface
ADEOS - Adaptive Domain Environment for Operating Systems
SMP - Symmetric Multi Processor
RT - Real Time
RTOS - Real Time Operating System
GPOS - General Purpose Operating System
IPC - Inter Process Communication
FIFO - First In First Out
LIFO - Last In First Out
RTHAL - Real Time Hardware Abstraction Layer
POSIX - Portable Operating System Interface
API - Application Programming Interface
srq - System ReQuest
SHM - Shared Memory
ioctl - Input Output ConTroL
fops - File OPerationS
sysctl - SYStem ConTroL
GPL - General Public License
ISR - Interrupt Service Routine
DSR - Deferred Service Routine
LXRT - LinuX RealTime

PSDD - Process Space Development Domain
PSC - POSIX Signaling Core
CPM - Communication Processor Module
SMI - System Management Interrupt
CNC - Computer Numeric Control
CAN - Control Area Network
MAC - Media Access Control
TCP - Transfer Control Protocol
UDP - User Datagram Protocol
IP - Internet Protocol
IPv4 - Internet Protocol version 4
HW - HardWare
NIC - Network Interface Card
RTSock - RealTime Sockets
QOS - Quality of Service
GUI - Graphical User Interface
LNET - Lightweight NETwork
CPU - Central Processing Unit
RAM - Random Access Memory
VM - Virtual Memory
VFS - Virtual FileSystem (Layer)
GNU - GNU not UNIX
OS - Operating System
PC - Personal Computer
PC/AT - Personal Computer / Advanced Technology
HZ - HertZ
LTT - Linux Trace Toolkit
FT - Fault Tolerant
DMA - Direct Memory Access
GDB - GNU Debugger
L2 - Level 2
P4 - Pentium 4
SBC - Single Board Computer
PIII - Pentium 3
ICACHE - Instruction CACHE
DCACHE - Data CACHE
DIDMA - Double Indexed Dynamic Memory Allocator
GFP_ - GetFreePage_ (i.e. GFP_KERNEL)
EDF - Earliest Deadline First
RM - Rate Monotonic
RMA - Rate Monotonic Algorithm
SRP - Stack Resource Protocol
CSP - Ceiling Semaphore Protocol
ISA - Industrial Standard Architecture

IRQ - Interrupt ReQuest
IEEE - Institute for Electrical and Electronics Engineers
IDE - Integrated Device Electronics
PCI - Peripheral Component Interconnect
Tcl/TK - Terminal Control Language / ToolKit
IPI - Inter Process Interrupt
IO - Input/Output
pid - Process IDentifier
NMT - New Mexico Tech, University of New Mexico
IPC - Inter Process Communication
TLSF - Two Level Segregated Fit
CSMA - Carrier Sense Multiple Access
CSMA/CD - Carrier Sense Multiple Access/ Collision Detection
CSMA/CA - Carrier Sense Multiple Access/ Collision Avoidance

Bibliography

[1] Gary Nut:

[2] Michael Barabanov: *RTLinux*, 1996, New Mexico Tech.

[3] Alessandro Rubini, Jonathan Corbet: *Linux Device Drivers, 2nd Edition*, O'Reilly, 2001, ISBN 0-59600-008-1

[4] Borko Furht et. al.: *Real Time UNIX Systems, Design and Application Guide*, KAP, 1991, ISBN 0-7923-9099-7

[5] OpenTech: *RTLinux Cache optimization*, 2004,

[6] Linux Kernel Web Resource, *http://www.kernel.org*

[7] Daniel P Bovet, Marco Cesati: *Understanding the Linux Kernel*, O'Reilly, 2003, ISBN 0-596-00213-0

[8] High Resolution POSIX Timers, *http://sourceforge.net/projects/high-res-timers/*

[9] Will Dinkel, Douglas Niehaus, Michael Frisbie, Jacob Woltersdorf: *KURT-Linux User Manual*, University of Kansas,2002, *http://www.ittc.ku.edu/kurt*

[10] Utime Webresource $\longrightarrow$ UTIME = Micro-Second Resolution Timers for Linux: *http://www.ittc.ku.edu/utime/*

[11] Borko Fuhrt, Dan Grostick, David Gluch, Guy Rabbat, John Parker, Meg McRoberts: *Real-Time UNIX Systems - Design and Application Guide*, KAP, 1991, ISBN 0-7923-9009-7

[12] MontaVista Annoucement: *Design of a Fully Preemptable Linux Kernel: http://www.linuxdevices.com/news/NS7572420206.html*

[13] Montavista Download Page for Preview Kits, *http://www.mvista.com/previewkit/index.html*

[14] Kevin Morgan: *Preemptible Linux: A Reality Check*, MontaVista White Paper, 2001

[15] Joachim Nilssson, Daniel Rytterlund: *Modular Scheduling in Real-Time Linux*, Department of Computer Engineering, Mälardalen University, December 3, 2000

[16] Clark Williams: *Linux Scheduler Latency*, March 2002, *http://www.linuxdevices.com/articles/AT8906594941.html*

[17] Dave Phillips: *Low Latency in the Linux Kernel*, November 2000. *http:/www.oreillynet.com/pub/a/linux/2000/11/17/low_latency.html*

[18] Low Larency Patch Web Resource,*http://www.zip.com.au/~akpm/linux/schedlat.html#downloads*

[19] *http://www.linuxjournal.com/article.php?sid=6405*

[20] Mel Gorman: *Understanding the Linux Virtual Memory Manager*, July 2003, *http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/*

[21] J.P. Lehoczky, L. Sha, J. K. Strosnider, H. Tokuda: *Fixed Priority Scheduling Theory for Hard Real-Time systems*, 1991, CMU Pitsburg, published in the `Foundations of Real-Time Computing`, Kluwer Academic Publishers.

[22] C.L. Liu, J.W. Layland: *Scheduling algorithms for multiprogramming in a hard real-time environment*, JACM, 20, 1973

[23] P. Balbastre, I. Ripoll: *Integrated Dynamic Priority Scheduler for RTLinux*, University of Valencia (DISCA), 2002.

[24] oleg@nmt.edu, {monotonic2.0.29}, *ftp://ftp.rtlinux.at/pub/rtlinux/contrib/applications/monotonic/monotonic2.0.29.tar.gz*

[25] T.B. Backer: *Stack-Based Scheduling of Real-Time Processes*, Journal of Real-Time Systems

[26] J. Vidal, F. Gonzalves, I. Ripoll: *POSIX TIMERS implementation in RTLinux, RTLinux-3.2-pre3, http://www.rtlinux-gpl.org*

[27] V. Yodaiken: *Priority inheritance is a non-solution to the wrong probem*, Technical report, FSMLabs Inc., 2002

[28] J. P. Lehocky, L. Sha, J. K. Strosnider: *Enhanced aperiodic responsivness in hard real-time environments*, Proc. 8^{th} IEEE-RTSS, 1987

[29] [RTAI API Documentation] E. Bianchi, L. Dozio, P. Mantegazza: *A Hard Real Time support for LINUX*, DIAPM Politectnico di Milan, 2003

[30] [Single UNIX Specification Version 2]

[31] [POSIX SCHED_FIFO] *http://www.opengroup.org/onlinepubs/007908799/xsh/realtime.html#tag_000_008_004_000*

[32] [RTAI overview] *http://www.schwebel.de/authoring/elektronik-rtai.pdf*

[33] Andrew S. Tanenbaum: *Computer Networks,* 3^{rd} Edition, Prentice-Hall, 1996, ISBN 0-13-394248-1

[34] Herman Kopetz: *Real-Time Systems - Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7

[35] Dietmar Dietrich, Wolfgang Kastner, Thilo Sauter: *EIB Gebaeudebussystem, Huethig Verlag Heidelberg*, 2000, ISBN 3-7785-2795-9

[36] Andrew S. Tanenbaum: *Modern Operating System,* (2^{nd} Edition), Prentice-Hall, 2001, ISBN 3-7785-2795-9

[37] G. H. Alt,R. S. Guerra, W. F. Lages: *An assessment of real-time robot control over IP networks*, Proceeding of the 4th RTLinux WorkShop, Federal University of Rio Grande do Sul, Electrical Engineering Department, Porto Alegre Brazil

[38] [Hirarchical Token Buffer] *http://luxik.cdi.cz/~devik/qos/htb/*

[39] [Diffserv field marker] *http://www.gta.ufrj.br/diffserv/*

[40] [Packet classifier API] *http://icawww1.epfl.ch/linux-diffserv/*

[41] [Real Time Message Passing Interface] *http://www.mpirt.org*

[42] [Linux QOS Library] *http://www.coverfire.com/lql/*

[43] [Linux QOS Page] *http://qos.ittc.ku.edu/*

[44] [Linux Diffserve] *http://www.opalsoft.net/qos/DS.htm*

[45] [BootPrompt-HOWTO] *http://*

[46] [LTT for RTAI] K. Yaghmour: *Monitoring and Analyzing RTAI System Behavior Using the Linux Trace Toolkit*, Proceedings of the 2nd Real Time Linux Workshop, Orlando, 2000.

[47] [POSIX SHM and FIFOs] C. Dougan, M. Sherer, *RTLinux POSIX API for IO on Real-time FIFOs and Shared Memory*, FSMLabs Inc, 2003

[48] [RT-Synchronisatoin] V. Yodaiken: *Temporal inventory and real-time synchronisation in RTLinux/Pro*, FSMLabs Inc., 2003

[49] [EMBEDIX Programing Guide] *Embedix Realtime Programming Guide 1.01*, Lineo Inc., 2001

[50] [Proc Utilities] N. Mc Guire, */proc based Utilities for Embedded Systems*, OpenTech, 2003

[51] [Using /proc] N. Mc Guire, *Proc Filesystem for Embedded Linux - Concepts and Programming*, OpenTech, 2003

[52] [Linux Kernel Programierung] M. Beck, H Boehme, M. Dziakdzka, U. Kunitz, R. Magnus, C. Schroter, D. Verworner, *Linux Kernel-programmierung, Algorithmen und Strukturen der Version 2.4*, Addison-Weley, 2001

[53] [RTLinux/GPL] *http://www.rtlinux-gpl.org/*

[54] [RTL Kernel resources] *http://www.rtlinux-gpl.org/rtlinux-3.2-pre3/example/kernel_resources/*

[55] [Kernel Resources] Nicholas Mc Guire, Using Linux Kernel Facilities from RT-threads *http://www.realtimelinuxfoundation.org/events/events.html*

[56] [Stodolsky Fast IRQ] Daniel Stodolsky, Brian N. Bernshaw, *Fast Interrupt Priority Managment in Operating System Kernels*,CMU and WU, 1993

[57] [Comedi] *http://www.comedi.org/*