The ABCD Proto-Kernel™ Guide

(Embedded Software Document)

Document Number C002274

2004/03/09

(C) 2002 CONNOTECH Experts-conseils inc.

Document Revision History

C-Number	Date	Explanation
C001299	2002/11/18	Very incomplete version -- a portion hidden for distribution
C001534	2003/10/17	First release with an overall coverage of the subject area
C002274	2004/03/09	ABCD Proto-Kernel version 1.2: improved the text in section 6.5.2, useful addition: timer (enhanced text) and low power mode, added text on maintenance commands, reference to lab-comm

Table of Contents

1. Introduction 7
- 1.1 The ABCD Proto-Kernel™ in a Nutshell 7
- 1.2 Target Microprocessor Family 7
- 1.3 ABCD "Proto"-Kernel Like "Protozoa," Not Necessarily "Prototype" 7
- 1.4 Document Organization 8
- 1.5 Related Documents 8
  - 1.5.1 Target Microprocessor Documents 8
    - 1.5.1.1 Core PowerPC Documents 8
    - 1.5.1.2 Motorola MPC8XX Microprocessor Documents 9
  - 1.5.2 PPCMB/850 Target Microprocessor Module Documents 9
  - 1.5.3 Related Software Documents 9
  - 1.5.4 Other Documents 10
- 1.6 Challenges of Embedded Software 10
- 1.7 Notation Used in this Document 11

2. Basic ABCD Concepts 11
- 2.1 'A' - Interrupt Dispatching 11
- 2.2 'B' - Fixed Priority Scheduling 12
- 2.3 'C' - Mutual-Exclusion Semaphores 12
- 2.4 'D' - A Queuing Mechanism 12

3. The ABCD Proto-Kernel™ Reference 13
- 3.1 Programming Environment Characteristics 13
- 3.2 'A' - Interrupt Dispatching 14
  - 3.2.1 Unified Naming of Interrupt Sources 15
  - 3.2.2 Interrupt Sources Masking and Temporary Disabling 15
  - 3.2.3 Interrupt Service Routines Registration 18
    - 3.2.3.1 Interrupt Service Routines Written in Assembler 19
    - 3.2.3.2 Interrupt Service Routines Written in C or C++ 20
  - 3.2.4 Interrupt Service Routine Software Context 20
  - 3.2.5 Interrupt Service Routine Epilogue 20
- 3.3 'B' - Fixed Priority Scheduling 21
  - 3.3.1 Overview of Tasks and Scheduling 21
  - 3.3.2 Scheduler Operation 21
  - 3.3.3 Tasks Priorities Declaration 24
  - 3.3.4 Execution Context Identification 25
  - 3.3.5 Task Local Data 25
- 3.4 'C' - Mutual-Exclusion Semaphores 26

3.4.1 Overview 26
3.4.2 Usage Rules 26
3.4.3 The Mutex Deadlock Prevention Mechanism 27
3.4.4 Implementation Issues 28
3.5 'D' - A Queuing Mechanism 28
3.5.1 Overview 28
3.5.1.1 A Simple Memory-Based FIFO Mechanism 28
3.5.1.2 Embedded Software Refinements 29
3.5.1.2.1 Task Blocking on One or Two Queues 29
3.5.1.2.2 Support of Event Insertion by Interrupt Service Routines 29
3.5.1.2.3 Support of Multiple and Potentially Overlapping Insertion Operations 29
3.5.1.2.4 Preventive Programming Support for Queue Full Conditions 30
3.5.1.2.5 Event Queues Devoid of Queue Data Memory 30
3.5.1.3 ABCD Event Queue Usage Sample Code 30
3.5.1.3.1 Type and Object Declarations and Definitions 30
3.5.1.3.2 Queue Insertion Operation in a Task Context 32
3.5.1.3.3 Queue Insertion Operation in an ISR Written in C or C++ 34
3.5.1.3.4 Queue Insertion Operation in an ISR Written in Assembler 34
3.5.1.3.5 Queue Extraction Operation 35
3.5.2 Event Queue Declaration and Initialization 36
3.5.2.1 Usual Event Queue Declarations and Definitions 37
3.5.2.2 Event Queue Definitions for Queues Devoid of Entry Memory 38
3.5.3 Queue Entry Insertion Functions 39
3.5.3.1 Queue Entry Insertion by Tasks 40
3.5.3.2 Queue Entry Insertion by Interrupt Service Routines 41
3.5.3.3 Explicit Kernel Scheduling Invocation After Entry Insertion 41
3.5.3.4 Queue Full Condition 41
3.5.4 Queue Entry Extraction 42
3.5.4.1 Queue Entry Extraction Functions 43
3.5.4.2 Queue Empty Condition 43
3.5.4.3 Event Queue Wait Functions 43

4. Strategies for Mission-Critical Interrupt Latency 44
4.1 Support for Delicate Balance of Priorities for to Achieve Adequate ISR Latencies 44

4.1.1 Introduction 44
4.1.2 The First Approximation Requirement for Nesting of ISRs 44
4.1.3 Better than Nested ISRs, a Critical Real-Time Arena 45
4.1.4 Conclusion 46

5. Useful Additions to the ABCD Proto-Kernel™ 47
5.1 Buddy System Memory Allocation 47
5.2 Timer Facilities 48
5.2.1 Standard Type clock_t Support 48
5.2.2 General-Purpose Timers 48
5.2.2.1 Blocking Wait Function 49
5.2.2.2 Wait Functions for Queued Timer Expiry Events 49
5.2.3 System Tick Timer 50
5.3 The FlashCnl Software Library 51
5.4 Embedded Interactive Monitor Facilities 51
5.4.1 Traces Through a Communications Channel 51
5.4.2 Maintenance Commands 52
5.5 Software Diagnostics Support 53
5.5.1 Software Diagnostics Using LEDs 54
5.5.2 Software Diagnostics Using Service History Data Recording 55
5.5.2.1 Basic Facilities for Service History Recording 55
5.5.2.2 Processor Exceptions or Traps 55
5.5.2.3 Simple Programmed Software Crash 55

6. Implementation on the Motorola MPC8xx Processor Family 55
6.1 Interrupt Dispatching with the Motorola MPC8xx 56
6.1.1 MPC850 Interrupt Source Hierarchy 56
6.1.2 Names Assigned to Regular Interrupt Sources 57
6.1.3 Names Associated with Software Extension of Interrupt Controller Logic 59
6.1.4 A Singular Synchronization Requirement 62
6.2 Assembler Language ISRs Programming 62
6.2.1 ISR Entry and Epilogue 62
6.2.1.1 ISR Entry Conditions 63
6.2.1.2 ISR Epilogue 65
6.2.2 Services to ISRs Written in Assembler 66
6.2.2.1 Event Queue Insertion by ISRs Written in Assembler 66
6.2.2.2 Interrupt Masking Service 66
6.2.2.3 Timestamping Service in ISRs Written in Assembler 66
6.3 CPU Context Save Areas 66
6.3.1 Task Interrupted by an Asynchronous Event 67

6.3.2 Task Interrupted Synchronously . . . 67
6.4 Allocation of Resources . . . 67
6.4.1 CPU Registers . . . 67
6.4.2 Machine State Register Bits . . . 68
6.4.3 PowerPC Exception Vectors . . . 68
6.5 Communications Processor . . . 69
6.5.1 Communications Processor Configuration . . . 70
6.5.2 Handling of Buffer Descriptors . . . 71
6.5.3 Naming Conventions . . . 73
6.6 Low Power Management . . . 74
6.6.1 Idle CPU Power Saving . . . 75
6.6.2 System Sleep Power Saving . . . 75

Annex A. Partial List of Acronyms . . . 76

Annex B. Typical Embedded Software Application Design . . . 78
B.1 Overview of the Guardian Automated Vehicle System . . . 78
B.2 Preliminary Hardware Specifications . . . 79
B.3 Embedded System Software Organization . . . 80
B.3.1 Typical ABCD Proto-Kernel™ Task Organization . . . 80
B.4 List of tasks . . . 82
B.4.1 High Priority Task Group . . . 82
B.4.1.1 System Error Monitor Task . . . 82
B.4.1.2 Motor Control Loop for the Guardian Motion . . . 83
B.4.1.3 Infrared Sensor Data Acquisition Task . . . 83
B.4.2 Intermediate Priority Task Group . . . 83
B.4.2.1 Instantaneous speed, turn, and elevation monitoring task. . . . 84
B.4.2.2 Radio Receiver Communications Task . . . 84
B.4.2.3 Alarms Computation Task . . . 85
B.4.2.4 Radio Transmitter Communications Task . . . 85
B.4.3 Residual Primarily I/O Bound Tasks . . . 90
B.4.3.1 Miscellaneous Data Collection Task . . . 91
B.4.4 CPU Bound Tasks . . . 91
B.4.4.1 Route Navigation Task . . . 91
B.4.4.2 Data Synthesis Task . . . 91
B.4.4.3 Idle task . . . 91

List of figures

Figure 1. Interactions Between Tasks, Scheduler and ISR 23
Figure 2. ABCD Proto-Kernel™ Queue Classes Hierarchy 37
Figure 3. The Guardian Automated Vehicle Project 79

1. Introduction

1.1 The ABCD Proto-Kernel™ in a Nutshell

The ABCD Proto-Kernel™ is a software for deeply embedded systems. Such systems use a micro-controller or microprocessor to implement electronic monitoring and/or control of a device, but do not otherwise look like a computer.

The ABCD Proto-Kernel™ provides an elegant solution to the recurring problem of providing a "real-time operating system" for embedded projects small enough that the need for an operating system might be questioned in the first place. This solution is as simple as ABCD, respectively for a) interrupt dispatching, b) fixed priority scheduling, c) mutual-exclusion semaphores, and d) a queuing mechanism. This is a coherent selection of a minimal subset of the myriad of programming facilities provided by full-blown real-time operating systems.

The ABCD Proto-Kernel™ concepts has been field-tested in an actual embedded project for an innovative product with a lot of electronics integration and moderately complex control functions, where the decision to build a kernel, instead of buying one, was influenced by the need for software certification.

1.2 Target Microprocessor Family

The ABCD Proto-Kernel™ is designed with sophisticated microprocessors in mind. The first implementation targets the PowerPC, and more specifically, the Motorola MPC8xx processor family. Even if the ABCD Proto-Kernel™ is defined and documented with some abstraction from the actual implementation, references to the MPC8xx processor family are sometimes unavoidable.

The specific hardware product targeted by the ABCD Proto-Kernel™ implementation is the PPCMB/850 processor board. See the document [PPCMB850_HW_UG].

1.3 ABCD "Proto"-Kernel Like "Protozoa," Not Necessarily "Prototype"

Protozoa are "very small and simple *living things* found in *water*" and the ABCD Proto-Kernel™ versions are "very small and simple *functional operating systems* found in *devices*." Protozoa species and ABCD Proto-Kernel™ versions share all the essential characteristics of their respective advanced counterparts (plants and animals for protozoa, overly complex real-time operating systems for the ABCD Proto-Kernel™).

After millions of years of evolution, the protozoa are still present on earth among the advanced

life forms. The ABCD Proto-Kernel™ exhibits this capability to last, due to the right fit to the real-time challenges at the very beginning.

A prototype usually disappears when turned into a production product. The ABCD Proto-Kernel™ is not designed as the lasting foundation of deeply embedded systems.

1.4 Document Organization

This document introduces the ABCD concepts in the next section, using a tutorial perspective. The section TBD describes a typical embedded application software using the ABCD Proto-Kernel™. The proof of the pudding is in the eating, as the saying goes, so most readers should look at this TBD document section describing how the ABCD Proto-Kernel™ is used in practice. A more formal description of the ABCD Proto-Kernel™ is contained in section TBD.

The usual measure of performance for an embedded system is its ability to meet strict deadlines in its data acquisition and control functions. This boils down to the system reaction time to external events triggering interrupt signals. Accordingly, a section of this document provides strategies for mission-critical interrupt latency. This is an area where the ABCD Proto-Kernel™ implementation details make a significant contribution to solving non-trivial requirements.

A last section of this document covers a number of additions to the core ABCD Proto-Kernel™ described so far. The presence of such additions is not an indication that ABCD would fall short of meeting the kernel requirements of typical embedded systems. None of these additions alter or extend the fundamental control of the processor and software effected by the ABCD Proto-Kernel™.

1.5 Related Documents

1.5.1 Target Microprocessor Documents

1.5.1.1 Core PowerPC Documents

[PPC_ARCH32]

Motorola, *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture*, document MPCFPE32B/AD, 12/2001, revision 2

[PPC_ARCH32_ERRATA]

Motorola, *Errata to Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture, Rev. 2*, document MPCFPE32BAD/AD, revision 0, 10/2002

[MPC8XX_INSTR]

Motorola, *MPCxxx Instruction Set*

[PPC_SIMPL_MNEMONICS]
Motorola, *Simplified Mnemonics for PowerPC™ Instructions*, Application note AN2491, Rev. 0, 9/2003

[PPC_ABI]
Steve Zucker, SunSoft, and Kari Karhi, IBM, *System V Application Binary Interface, PowerPC Processor Supplement*, September 1995

[PPC_EABI]
Stephen Sobek, Motorola, and Kevin Burke, IBM, *PowerPC Embedded Application Binary Interface*, 1995/01/10

1.5.1.2 Motorola MPC8XX Microprocessor Documents

[MPC850_UG]
Motorola, *MPC850 Family User's Manual, Integrated Communications Microprocessor*, document MPC850UM/D, Rev. 1, 1/2001

[MPC850_UGERRATA]
Motorola, *MPC850 Communications Microprocessor User's Manual Errata*, document MPC850UMAD/D, Rev. 0.3, 6/2002

1.5.2 PPCMB/850 Target Microprocessor Module Documents

[PPCMB850_HW_UG]
CONNOTECH Experts-conseils inc., *PPCMB/850 Hardware User's Guide*, Document Number C001954, 2003/09/20

1.5.3 Related Software Documents

[ABCD_LINK_N_LOAD]
CONNOTECH Experts-conseils inc., *ABCD Proto-Kernel™ Software Link and Load Process (Embedded Software Document)*, PPCMB/850 Product Family Documentation, Document Number C002468, 2004/03/09

[PPCMB850_INIT]
CONNOTECH Experts-conseils inc., *PPCMB/850 Initialization Sequence, (Embedded Software Document)*, Document Number C001804, 2003/09/29

[FLASHCNL]
CONNOTECH Experts-conseils inc., *The FlashCnL API, A Flash Memory Configuration and Log Application Programming Interface*, Document Number C001270, 2003/06/17

[LAB_COMM_UG]
CONNOTECH Experts-conseils inc., The ABCD Proto-Kernel Networking Specifications, including the LAB-COM Utility Guide, Document Number C002424, 2004/03/09

1.5.4 Other Documents

[RFC1549]
Network Working Group, *PPP in HDLC Framing*, Internet Request for Comments 1549, RFC1549, December 1993

1.6 Challenges of Embedded Software

Generally speaking, embedded software programming is about using adequate software synchronization mechanisms in order for the system to behave as intended even in the most unusual or improbable sequence of events. Throughout this document, examples are given to illustrate how the ABCD Proto-Kernel™ synchronization mechanisms might be used. In most cases, if not all, those examples are just one way of achieving adequate synchronization and the alternate means are adequate as well.

Embedded systems are sometimes used in critical applications. The application fields where embedded software can be critical include medical devices, nuclear, aerospace, and information security. In each of these fields, more or less formal embedded software certification activities may be required before the embedded software can be put in operation.

The state-of-the-art practice in the field of software certification comes from the civil aviation application area with the RTCA DO-178B/ED-12B standard document. This standard was issued in the US as document RTCA DO-178B (Requirements and Technical Concepts for Aviation document) and in Europe as document EUROCAE ED-12B (European Organisation for Civil Aviation Electronics). The language of DO-178B is fairly esoteric for an outsider and the document RTCA DO-248B, Final Annual Report For Clarification Of DO-178B “Software Considerations In Airborne Systems And Equipment Certification” is a recommended companion document.

At some point, software certification for critical embedded systems becomes a political process. For the manufacturer, it represents an entry ticket to the market, and a barrier to entry against potential competitors. Moreover, participants have an incentive to avoid liability exposure, with a

corresponding inclination to ignore a challenging technical issue when feasible. For the certification authority personnel, the safety consciousness is not always distinct from paranoia. But paranoia is seldom a suitable state of mind for rational thinking and effective use of available information.

In this context, the *GNU-style free software* is a welcome initiative: the effectiveness of the liability limitation clause is supported by the absence of business relation between the collaborative software *development activity* and the unrestricted *software use*. In this context, the individual developers can openly address the safety-related technical issues, without undue fear of exposing the software development organization to excessive potential liability.

The ABCD Proto-Kernel™ developers, either the original developer or the contributors in GNU-style free software licensing, shall not be held liable for any damage resulting from the use of the ABCD Proto-Kernel™ in critical applications. Nonetheless, the ABCD Proto-Kernel™ design, development and recommended usage are somehow influenced by the criteria generally used to base a software certification decision.

1.7 Notation Used in this Document

The ABCD Proto-Kernel™ uses mainly the C/C++ language for its implementation, with liberal use of preprocessor symbols. Moreover, the ABCD Proto-Kernel™ is intended primarily for the Motorola MPC8xx processor family, and the MPC850 User's Manual is an important reference document in many sections of the present document (references [MPC850_UG] and [MPC850_UGERRATA]). The following fonts are used

`unsigned long v_temp;` is used for C/C++ language source code,

SYMBOL_XYZ is used for preprocessor symbols,

source.h is used for file names.

SIUMCR(FRC) is used when referring to a MPC8xx register or a specific field in a MPC8xx register.

2. Basic ABCD Concepts

2.1 'A' - Interrupt Dispatching

A basic function of any operating system or kernel is to handle the microprocessor interrupts. This is usually closely tied to the microprocessor architecture. The ABCD Proto-Kernel™

provides a unified handling of interrupts that logically dispatch the interrupt signals to the interrupt service routine (ISR) that correspond to the exact source of the interrupt, as if the microprocessor had a large number of independent interrupt vectors. In doing so, the ABCD Proto-Kernel™ prepares the ISR execution context (some limitations apply to the software statements that make up the ISR). Lastly, the interrupt handling includes ISR epilogue facilities that restore a normal software task context once the ISR processing is complete.

It is the application developers' responsibility to code the interrupt services routines required by the application, to register them with the ABCD Proto-Kernel™ interrupt dispatching, and to request the enabling of the interrupt source associated with each ISR.

2.2 'B' - Fixed Priority Scheduling

Any operating system or kernel provides multi-tasking of some sort. The ABCD Proto-Kernel™ scheduling is a simple control program that supports multi-tasking with fixed priority and no time-slicing. This scheduling ensures that the highest priority task that is not blocked is indeed allowed to run until it preforms a blocking action. A task is blocked when it waits for an event to be signaled in a queue (the 'D' portion of 'ABCD') or for a mutual exclusion semaphore to be relinquished (the 'C' portion of 'ABCD').

Except for the ISRs, the application software is known by the ABCD Proto-Kernel™ as a fixed list of tasks, with a priority ordering. It is the application programmers' responsibility to code the tasks with appropriate blocking actions, so that the system functions are performed as intended.

2.3 'C' - Mutual-Exclusion Semaphores

Mutual exclusion semaphores (mutexes) are a classical operating system textbook concept. They provide a guarantee that task scheduling will never induce occasional interfere between two or more tasks that must access a resource in turns. The ABCD Proto-Kernel™ mutex implementation includes the basic facilities of mutual exclusion semaphores, plus an ordering facility that enforces a discipline in the mutex reservation order (thus preventing potential deadlocks).

It is the application developers' responsibility to determine where mutexes should be used for orderly access to the application resources by multiple tasks. The definition of resources is strictly an application issue (e.g. a set of global variables that must remain coherent such as a x,y,z vector for a three dimensional position). Likewise, the association of a resource to a semaphore is also an application issue.

2.4 'D' - A Queuing Mechanism

The queuing mechanism contributes a lot to the ABCD Proto-Kernel™ potential. At the outset, a queue is a simple memory-based FIFO which tasks and ISRs can access freely because the microprocessor memory is neither protected nor managed by the kernel. The queuing mechanism is used with the scheduling facilities to provide the preferred way of blocking a task, and the preferred ISR-to-task signaling method (queues can always be used independently of scheduling). It is possible for multiple tasks and ISRs to insert entries into the same queue, without any special coding for synchronization. We recommend to use the C++ interface to the queuing mechanism because the C++ type checking detects more software coding errors at compile-time (sparing valuable target system testing time).

The application developers should study the details of the ABCD Proto-Kernel™ queuing mechanism to understand how it shapes the design of the application software. Typically, the design will encompass one or two 'event queues' (a minor abuse of language since an event queue is no different from any other queue) on which the task will block. The application software design will then define for each queue, the specific events that trigger queue entry insertions.

3. The ABCD Proto-Kernel™ Reference

3.1 Programming Environment Characteristics

The ABCD Proto-Kernel™ software is predominantly coded in the C/C++ high level programming language, assembler source code being limited to cases where C constructs are hardly applicable (e.g. processor context switch around ISRs and before and after the task scheduling logic). Implicitly, the ABCD Proto-Kernel™ supports the view that at most a very small portion of any software product should be coded in assembler.

The ABCD Proto-Kernel™ software is compiled as C++ source code but its 'object orientation' is fairly low (this statement is intentionally vague as 'object orientation' is a highly debatable topic). As a significant exception, the queuing mechanism uses 'advanced' C++ language features, namely class derivation and template mechanism. The application programmer need not understand the many subtleties of the C++ programming language to use the kernel queues. However, a pure C source code strategy is incompatible with the ABCD Proto-Kernel™ as it stands.

The static C++ object constructors are called in the kernel initialization sequence before the highest priority task is started. Fo details, see the reference [PPCMB850_INIT].

The ABCD Proto-Kernel™ scheduling uses some specific "threading model" for "task local

data” (the task local data is piggy-backed to thread context data controlled by the task scheduling). Static pointer initialization to variables in the task local data memory is not allowed and may produces undefined behavior, but the programmer might receive no warning message if such pointer initialization occurs in the source code. See the discussion of task local data in the document [ABCD_LINK_N_LOAD].

The kernel software is provided in source code form and is intended to be linked with the application software in a single software image ready to be loaded in the target system. Along with this unified software image creation model,

- no use is made of the processor protection capabilities (that isolate the user application code from the supervisor or kernel code), and
- the target memory address space is assumed to be universally available, to kernel code, to application tasks, and to ISRs.

Moreover, this assumes the absence of a virtual memory scheme, hence little or no use of the processor MMU (Memory Management Unit) capabilities.

However, the performance of modern processors may be critically dependent on the memory cache, and this may lead to memory usage restrictions in the case of the data cache. When the data cache is enabled for a section of the memory, DMA (direct memory access) activity to or from this region may lead to data inconsistencies. In order to allow the use of the processor data cache where it may noticeably impact the software performance, the following restriction is specified:

> The embedded application shall not trigger DMA activity to or from the task stack space (local variables in C/C++) and the task local data space.

When considering the potential application bugs in the unified kernel-application software model, embedded software development may appear like going out on a limb. Indeed, not relying on processor protection features calls for a greater programming discipline. However, the alternate approach is to use a kernel with protection features that prevent tasks from unrestricted cooperation, kernel-defined inter-task communications facilities, and a restrictive device driver model for ISRs. This larger set of kernel concepts represents a more tedious software coding exercise for an end-result that should contain pretty much equivalent application tasks interactions. In either case, an application task failure or an ISR misbehavior is just as likely to result in a system failure from an operational perspective. Experience with the unified software model teaches that it is indeed an acceptable solution (simplicity pays for itself).

3.2 ‘A’ - Interrupt Dispatching

The ABCD Proto-Kernel™ handles most of the details of interrupt management according to the processor rules, yet the application remains in control of which interrupt signal are handled, and how.

3.2.1 Unified Naming of Interrupt Sources

The ABCD Proto-Kernel™ presents a unified interface to the interrupt sources. The naming/numbering of interrupt sources is made to accommodate up to 63 interrupt sources.

With each defined ISR, there are three associated C preprocessor symbols. Given an interrupt signal XYZ, then the associated symbols are

ABCD_ISRDISP_XYZ
: Uniquely refers to the interrupt source XYZ for ISR dispatch management purposes. This is the index to use to register an ISR for the interrupt source XYZ.

ABCD_ISRMASK_XYZ_H and ABCD_ISRMASK_XYZ_L
: Refer to a bit for masking interrupt source XYZ. These are a pair of preprocessor symbols, respectively for the high and low portions of a 64 bits value representing a bit mask for enabling or disabling the interrupt source XYZ (one of the two symbols is zero while the other is an exact power or two).

When an interrupt source is non-maskable but nonetheless managed by the ABCD Proto-Kernel™, the symbols ABCD_ISRMASK_XYZ_H and ABCD_ISRMASK_XYZ_L are not defined.

In some cases, interrupt dispatching may be provided for events that are grouped for masking purposes. If the source XYZ is grouped with other sources under the event name GRP, the symbols ABCD_ISRMASK_XYZ_H and ABCD_ISRMASK_XYZ_L are not defined and the symbols ABCD_ISRMASK_GRP_H and ABCD_ISRMASK_GRP_L must be used for source masking purposes. The symbol ABCD_ISRDISP_GRP is reserved for ABCD Proto-Kernel™ use.

The ABCD Proto-Kernel™ software makes up for the diversified hardware architecture found in the MPC850 and presents a unified ISR management interface. The section 6.1 on page 56 gives the specifications for the MPC850 implementation.

3.2.2 Interrupt Sources Masking and Temporary Disabling

The ABCD Proto-Kernel™ provides functions for selectively disabling and enabling interrupts. It is recommended that application code disable only the interrupt sources which are relevant in each context where a temporary disabling of interrupts is desired. As an example, if an application code wishes to read a few variables related to an input device which is sampled in a ISR triggered by a timer interrupt source, it needs only to disable this specific timer interrupt source while copying the variables to an application buffer (actually the queue mechanism is usually a better synchronization strategy between ISR and tasks, but interrupt source masking remains an option). In order to do so, the application uses a pair of related function calls to disable some interrupt sources, and later restores the interrupt context to what it was in the first place. An application provided memory area is used to store the context between the two calls.

The same functions used for temporarily disabling interrupts allows the temporary preventing of kernel application scheduling, a feature sometimes called "critical section" in the documentation of other kernels and operating systems. As a synchronization mechanism, the temporary prevention of kernel scheduling freezes other applications which might interfere with the current one, but lets the enabled ISRs do their job. Any scheduling request by an ISR is delayed to when the current application calls the interrupt restore function (re-)allowing the kernel scheduling. The kernel prevention feature of interrupt masking management functions should be used sparingly because it disrupts the task priorities in a very crude way. It may find applications when a mutex declaration or application-specific interrupt masking is not practical.

The temporary storage used to hold the interrupt masking context is the following structure definition:

```
struct abcd_isr_masks_str
{
     unsigned long high_mask;
     unsigned long low_mask;
};
```

The three application-level functions to manage interrupt source masking are:

```
extern struct abcd_isr_masks_str abcd_int_enable
          ( unsigned long high_mask
          , unsigned long low_mask);
```

The function **abcd_int_enable** enables some interrupt sources in the current interrupt masking context. It can not disable anything. It returns the interrupt masking context prior to the establishment of the updated context.

```
extern struct abcd_isr_masks_str abcd_int_disable
          ( unsigned long high_mask
          , unsigned long low_mask);
```

The function **abcd_int_disable** can disable specific interrupt sources in the current interrupt masking context and/or request the prevention of kernel scheduling. Its return value contains the required arguments for a subsequent call to **abcd_int_restore** (see below), which should occur in the same source code block because disabling interrupts is usually a temporary operation. Kernel preventing is requested by bitwise OR'ing the preprocessor symbol ABCD_KERNEL_CTX with the **high_mask** argument to the call to **abcd_int_disable**.

```
extern struct abcd_isr_masks_str abcd_int_restore
          ( unsigned long high_mask
          , unsigned long low_mask);
```

The function **abcd_int_restore** is provided to restore the interrupt masking context to what it was before a prior call to **abcd_int_disable**. It returns the interrupt masking context prior to the call. A table below formalizes the effect of **abcd_int_restore** if other tasks (or ISRs) change the interrupt masking context in parallel with the current task.

Notes: (A) The behavior depicted in the table is implemented by the function **abcd_int_disable** restricting its result value to the interrupt sources that were not already masked-out in the interrupt masking context.

(B) If doubtful system behavior detection is desired, the application should match the result value of call to **abcd_int_restore**, (say **recent**) with the initial argument to the call to **abcd_int_disable** (say **request**), and consider the expression

```
( ( recent.high_mask & request.high_mask
                       & ~ABCD_KERNEL_CTX ) |
( recent.low_mask & request.low_mask ) )
```

as an error indication.

Interrupt source status in the first place	Source disabling requested?	Interrupt source status immediately before restoring the context	Interrupt source status after restoring the context
disabled	Yes	disabled	disabled
disabled	Yes	enabled (note 1)	enabled
enabled	Yes	disabled	enabled
enabled	Yes	enabled (notes 1, 2)	enabled
any	No	disabled	disabled
any	No	enabled	enabled

Notes: 1) If the intervening interrupt enabling operation has been done by another task, the software design should be put into question because the interrupt disabling service was not actually provided to the current task.

2) If the intervening interrupt enabling operation has been done by the current task, the software coding might be redundant.

Kernel scheduling prevention may only be done using the interrupt disable function. Also, kernel scheduling allowing may only be done with the interrupt restore function.

Some preprocessor macros are provided to simplify the naming of interrupt sources in these calls:

```
#define ISR_MASK_1(A) ...
#define ISR_MASK_2(A,B) ...
#define ISR_MASK_3(A,B,C) ...
#define ISR_MASK_4(A,B,C,D) ...
#define ISR_MASK_5(A,B,C,D,E) ...
```

These macro provide the **high_mask** and **low_mask** parameters for the interrupt source names passed as arguments (they make up the symbols ABCD_ISRMASK_XYZ_H and ABCD_ISRMASK_XYZ_L from the interrupt source name XYZ). For instance, if a sequence of application code must be protected against possible impact of the three interrupt sources SCC2, TMR1, and IRQ3, and to prevent the kernel scheduling at the same time, the following code fragment is recommended:

```
{
	struct abcd_isr_masks_str isr_context;
	isr_context=abcd_int_disable
			( ABCD_KERNEL_CTX | ISR_MASK_3 (SCC2, TMR1, IRQ3) );

	// ... Protected sequence of code

	abcd_int_restore
			( isr_context.high_mask, isr_context.low_mask );
}
```

In the description above, the current interrupt masking context is not necessarily the interrupt sources that are actually enabled at a given moment:

a) *every* interrupt sources can be disabled because the CPU runs in the interrupt disabled state, which does not prevent the application from managing the specific interrupt sources masking with the above functions; and
b) the ABCD Proto-Kernel™ provides a mechanism called the critical real-time arena, which quietly masks selected interrupts sources while high priority tasks are running (see section 4.1.3 on page 45).

3.2.3 Interrupt Service Routines Registration

It is the application's responsibility to register an ISR before the triggering signal source is unmasked.

The application registers an ISR (Interrupt Service Routine) for an interrupt signal by giving
- its entry point, and

if the ISR is coded in the C or C++ language,
- a TRIGTSK_LEVEL argument explained in section 3.2.5 on page 20.

The registration can be made with function-like preprocessor macros, respectively the macro ABCD_REGISTER_ASM_ISR for an assembler coded ISR and the macro ABCD_REGISTER_CPP_ISR for a C++ coded ISR.

```
ABCD_REGISTER_ASM_ISR(ISR_INDEX,ENTRY_POINT)

ABCD_REGISTER_CPP_ISR(ISR_INDEX,ENTRY_POINT,TRIGTSK_LEVEL)
```

The ISR_INDEX argument to these macro is one of the ABCD_ISRDISP_... symbols defined in the interrupt source naming specification. The other arguments are described in the following two sub-sections.

The macros ABCD_REGISTER_ASM_ISR and ABCD_REGISTER_CPP_ISR expands as assignment statements to the ABCD Proto-Kernel™ interrupt dispatch table, the variable named **abcd_isr_table** having a table entry type **struct abcd_isr_table_entry_str**.

It is sometimes reasonable for the application software design to add data members to the type definition **struct abcd_isr_table_entry_str**, e.g. if a single ISR is intended to process signals from various sources. The above macros and the preprocessor symbols of the form ABCD_ISRTABLE_... should be adapted according to changes made to the ISR dispatch table entry contents.

3.2.3.1 Interrupt Service Routines Written in Assembler

The ISRs coded in assembler are identified in the C++ source code by their entry point which is mimicked as a external C++ function taking no argument (name mangling comes into play here, see the development tools documentation about function overloading, mangling and demangling). This entry point name is passed to the macro ABCD_REGISTER_ASM_ISR as the argument ENTRY_POINT.

The implementation details applicable to the PowerPC implementation are given in section 6.2 on page 62.

3.2.3.2 Interrupt Service Routines Written in C or C++

When coded in C or C++, an ISR is a function declared as follows:

```
void isr_sample_name
     ( struct abcd_isr_table_entry_str *isr_entry );
```

As expected, the name **isr_sample_name** should be replaced by a different name for each different ISR. This name is passed to the macro ABCD_REGISTER_CPP_ISR as the argument ENTRY_POINT.

Note: The **isr_entry** parameter currently has no specific purpose but data member additions to the **struct abcd_isr_table_entry_str** could turn into a device indication, so that the same ISR function might handle interrupts from more than one similar sources.

The TRIGTSK_LEVEL argument to the macro ABCD_REGISTER_CPP_ISR is explained in section 3.2.5 on page 20.

3.2.4 Interrupt Service Routine Software Context

During execution of the ISR, the MSR[EE] bit is cleared (nested interrupts are not supported), and MSR[RI] is set (other processor exceptions are recoverable).

The execution environment for the ISRs written in C or C++ uses a dedicated stack and a dedicated task local data area. This is not the case for an ISR written in assembler, so it is invalid to call a C or C++ function from an ISR written in assembler.

Obviously, an ISR must not call any function that might block a task. This includes the mutex reservation operation (by definition, mutex reservation is not available to ISRs).

3.2.5 Interrupt Service Routine Epilogue

Upon exit, the ISRs can trigger one of three ISR epilogue sequences:

- Unconditional entry in the task scheduler (task level indication set to the SHRT_MAX constant defined in the standard header file <limits.h>).

- Entry in the task scheduler if the current task priority is less than a given task priority indication (task level indication set to a value in the range from ABCD_KERNEL_SCHED_CONTEXT_COUNT to the

ABCD_KERNEL_SCHED_CONTEXT_COUNT+ABCD_TASK_COUNT-1 constant inclusive). The priority indication should refer to the task that would be selected by the scheduler as a consequence of ISR execution, or a higher priority task. If so, the use of this alternative is merely an optimization opportunity.

Note: Here is the reasoning: if this ISR epilogue optimization is not used, the scheduler will more often be invoked to find out that the interrupted task is still the one that deserves to run. If used, the lower priority task will get its opportunity to run in the very same order of CPU time allocation, because the higher priority tasks will eventually relinquish the CPU, and only then the lower priority task will get its chance anyway.

- Unconditional return to the interrupted context (task level indication set to zero).

The ISR epilogue processing selection is made by a task level indication:

- for the ISRs written in assembler language, by specifying the argument TRIGTSK_LEVEL to the macro ABCD_ISR_EPILOGUE, or
- for the ISRs written in C or C++, at ISR registration time through the TRIGTSK_LEVEL argument to the macro ABCD_REGISTER_CPP_ISR.

In any event, an entry in the scheduler is deferred if the kernel scheduling is currently prevented, in which case a scheduler request flag is set so that the required scheduling will occur at the earliest opportunity.

3.3 'B' - Fixed Priority Scheduling

3.3.1 Overview of Tasks and Scheduling

A typical application based on the ABCD Proto-Kernel™ has a few specialized pooling loops, each implemented as a task. Each polling loop is notified of relevant external events by retrieving entries from one or two event queues, and processes the external events according to the details of each event queue entry. External events can be external ISR signals, serial communications line ISR, timer ISRs, and the like. Other tasks may insert entries in the task event queue(s); this is a basic task synchronization mechanism.

When there is no event to process for a polling loop, the CPU is allocated by the scheduler to a lower priority task. Each task in the ABCD Proto-Kernel™ is assigned a fixed and unique priority. The kernel scheduling operation is then defined as assigning the CPU to the highest priority task that is ready to run (the priority adjustment implied by the mutex logic applies to this context).

3.3.2 Scheduler Operation

The ABCD Proto-Kernel™ supports multiple tasks using a *kernel scheduler* which basically decides which task is to receive the CPU resource from now on until the next scheduler invocation. The kernel scheduler interaction with the tasks and the interrupt servicing is shown in figure 1. In this figure, the gray ellipses are the possible instantaneous CPU usages, and the arrows between them show the possible system transitions. The figure shows two possible CPU usages by one task (TASK RUNNING and TASK RUNNING, KERNEL SCHEDULING PREVENTED), but successive entries in the TASK RUNNING usage can be for a different task as decided by the scheduler. The bold and solid arrows illustrate the most typical transitions, and the thin dash-dotted arrows show the more exceptional ones.

Here is how the kernel scheduling is started. Once the system initialization code has set up the interrupt system and the internal state for the kernel, the initialization code is analogue to a task running, with kernel scheduling prevented. The first transition to occur is an entry into the scheduler.

Not shown in figure 1 is the system reaction to a software error detected by the processor as an exception (e.g. a division by zero). The ABCD Proto-Kernel™ does not support a crashed task. An event that would cause a task crash triggers a system crash. In this case, the kernel is simply stopped, interrupts are no longer serviced and an attempt is made to record and/or display the details of the system crash for troubleshooting purposes (see section 5.5.2.2 on page 55).

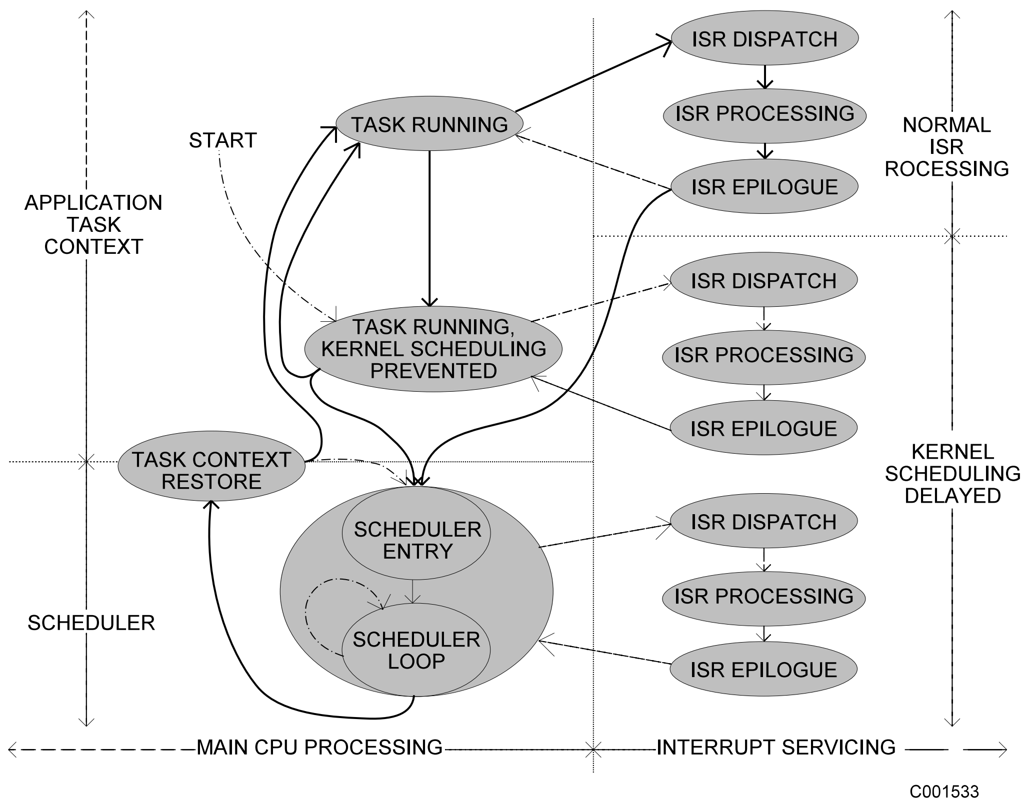

Figure 1. Interactions Between Tasks, Scheduler and ISR

For the kernel scheduler, a task can be in one of the following states:

- ready to run (**`abcd_kernel_sched_state_ready`**),
- waiting on one event queue (**`abcd_kernel_sched_state_one_queue_wait`**),
- waiting on either of two event queues (**`abcd_kernel_sched_state_two_queues_wait`**),
- waiting for a mutual exclusion semaphore (**`abcd_kernel_sched_state_mutex_wait`**).

As a general rule, the tasks in the ABCD Proto-Kernel™ have a fixed priority ordering. The only exception to this rule occurs when a task is waiting for a mutual exclusion semaphore, and the task that currently owns it has a lower priority. This lower priority is temporarily boosted to the priority of the waiting task. This priority adjustment is iterative in the case the current owner is itself waiting on another mutex.

The kernel scheduler source code is a simple loop which identifies the highest priority task that is ready to run, or for which the waiting condition is satisfied. This loop also implements the priority boost triggered by the mutual exclusion mechanism as described in the previous paragraph.

Some additional information on the kernel scheduler operation is found in section 3.5.3.3 on page 41.

A task has an associated task context state (contents of CPU registers). When a task is not running, its task context is saved and an indication tells whether it was interrupted by an asynchronous event (ISR preemption) or synchronously removed from the running state. See the section 6.3 on page 67 for the details on how the task context data is stored.

3.3.3 Tasks Priorities Declaration

The list of tasks of an application is supplied by the application software programmer in a C preprocessor include file named abcd_applic_tasklst.h. For each task in the system, there should be one call of either the C preprocessor macro ABCD_DECLARE_TASK or ABCD_DECLARE_TASK_WITH_MUTEXES that takes respectively three or five arguments:

1) the task number which is also the task priority (a higher numbered task has a higher priority), value range from ABCD_KERNEL_SCHED_CONTEXT_COUNT to the ABCD_KERNEL_SCHED_CONTEXT_COUNT+ABCD_TASK_COUNT-1 constant inclusive,
2) the name of the main function for this task (e.g. a C++ function **void a_task(void);**),
3) the word count for the stack to be allocated to this task,
4) the name of a global array of pointers to mutual exclusion semaphore objects (only for the macro ABCD_DECLARE_TASK_WITH_MUTEXES), and
5) the entry count for the preceding array (only for the macro ABCD_DECLARE_TASK_WITH_MUTEXES).

No double declarations are allowed for the same task number. It is not required to define every task. Tasks that are not defined in the abcd_applic_tasklst.h file are dummy tasks that wait forever on an event queue that never receives any event, except for the lowest priority task that is a default idle task.

When present in a task declaration, the fourth argument to the macro ABCD_DECLARE_TASK_WITH_MUTEXES (an array of pointers to mutual exclusion semaphore objects) gives the list of mutexes that will be initially reserved by the task. The number of such mutexes is given by the fifth argument to the macro ABCD_DECLARE_TASK_WITH_MUTEXES. A given mutex must not be so reserved by more than one task declaration.

When the abcd_applic_tasklst.h file is **#include**d by the application software building makefile

process in the compilation of one of the kernel source files, the configuration file abcd_incl.h (and the files **#include**d by abcd_incl.h) has been processed. The abcd_applic_tasklst.h file is **#include**d more than once. No native C/C++ declarations or statements should be present in this file. Preprocessor directives other than the macros ABCD_DECLARE_TASK and ABCD_DECLARE_TASK_WITH_MUTEXES may be used, provided they are compatible with the multiple inclusions of the abcd_applic_tasklst.h file.

3.3.4 Execution Context Identification

In addition to one execution context per task, there are two more execution contexts: one for the scheduler, and one for the ISRs written in the C or C++ language. The execution context identification has the main following uses:

for task scheduling purposes,

for task local data required by the run-time library,

for saving the processor context,

for ISR epilogue optimization (decision to omit scheduler entry) and ISR enabling optimization (the critical real-time arena feature).

The ISR execution context never saves processor context since the ABCD Proto-Kernel™ does not use any interrupt nesting capability of CPU interrupt controllers.

For to save the processor context, and for plain scheduling purposes, the variable **abcd_kernel_state.cur_priority** identifies the interrupted context for scheduling purposes, and the variable **abcd_current_task_index** identifies the kernel, task or ISR execution context for task local data purposes. Otherwise, these two variables hold the same value.

For ISR epilogue optimization and ISR enabling optimization, it is necessary to consider the current priority indication adjusted, as the case may be, by the mutex reservation logic, variable name **cur_priority_proxied**.

3.3.5 Task Local Data

When designing an embedded application software, there are very little justifications for the use of the task local data feature. It is unlikely that an application software requirement applies uniformly to every task in the embedded system. Actually, the task local data need originates from the early C run-time libraries - recent library additions seldom makes use of global statically

allocated variables.

If an application needs an object variable **x** of type **t** to behave independently for a few tasks T0, T1, ... Tn, various programming language facilities should be considered before the modification of the ABCD Proto-Kernel™ task local data implementation. One possibility is to use the declaration

```
t x[ ABCD_KERNEL_SCHED_CONTEXT_COUNT + ABCD_TASK_COUNT +
ABCD_KERNEL_ISR_CONTEXT_COUNT ];
```

and then the use of the task local copy of the object variable uses the notation

```
x[abcd_current_task_index]
```

If an application designer still wishes to use the task local data feature (e.g. there are compelling reasons to use **x** instead of **x[abcd_current_task_index]** in the above example), the initialization of the variables added to the task local data should be done in the following function

```
extern void abcd_task_local_data_init(void);
```

that is implicitly called immediately when the kernel scheduling starts, before every task entry point (the application programmer should modify the function **abcd_task_local_data_init**).

3.4 'C' - Mutual-Exclusion Semaphores

3.4.1 Overview

A mutual exclusion semaphore is a classical mechanism for synchronization of resource access by tasks in a kernel or operating system.

The logical link between a specific mutual exclusion semaphore and a given resource is not enforced by the ABCD Proto-Kernel™ software. For instance, if a mutual exclusion semaphore is intended to prevent concurrent update of some data structure S (e.g. S is used for centralized statistics gathering by a few tasks), it is the application software programmer responsibility to surround the source code sequence that updates S by a **mutex_reserve** / **mutex_relinquish** pair.

3.4.2 Usage Rules

The ABCD Proto-Kernel™ provides a simple mutual exclusion semaphore facility with the C++ type **class abcd_mutex_cl** and the member functions **mutex_reserve** and **mutex_relinquish** (defined in the file abcd_mutex.h). Nested reservation of the same mutex

by a given task is supported. Usually, the calls to **`mutex_reserve`** and **`mutex_relinquish`** must be paired for the application software to remain coherent in the long run.

It does not make sense to allocate a mutex object as a local variable. The mutex implementation relies on the static object constructor feature of the C++ language to initialize some control variables. Upon task startup, mutual exclusion semaphores are normally free, except for those that are listed as pre-reserved by the task declaration mechanism see section 3.3.3 on page 24.

The mutex reservation operation implies the possibility of a task switch (under the control of the kernel scheduler) if the mutex is currently owned by another task. For this reason, it makes little sense to reserve a mutex while the kernel scheduling is prevented, or when the interrupts are disabled at the CPU level (e.g. during the processor initialization sequence). Nonetheless, the mutex reservation/relinquish operation pairs can be part of low level application software functions which can run in a priori unforeseen contexts. Specifically, the mutex mechanism supports the mutex reservation/relinquish operation pair on a mutex while the kernel scheduling is prevented and/or when the interrupts are disabled at the CPU level, provided that the kernel scheduling remains so prevented without interruption between the reservation and the corresponding relinquish operation. If a mutex reservation can not be satisfied while the kernel scheduling is prevented, an error code is returned to the application software, *but the relinquish operation must nonetheless be performed.*

Note: This behavior of the ABCD Proto-Kernel™ is intended to support application software that do not handle unexpected error codes from calls to kernel service functions. The assumption is that it is less damaging for an ill-behaved application to intermittently corrupt one of its otherwise serialized resource than for the mutex mechanism to de-synchronize the nested mutex reservation scheme.

The mutex relinquishing operation can be done while the kernel scheduling is prevented and/or when the interrupts are disabled at the CPU level.

The mutex mechanism can not be reliably used by an ISR. If one or more tasks and one or more ISRs must serialize access to a resource, then the interrupt masking mechanism should be used instead of the mutex mechanism.

3.4.3 The Mutex Deadlock Prevention Mechanism

The mutex deadlock prevention feature makes it impossible for the textbook deadlock to occur (task A reserves mutex X, task B reserves mutex Y and attempts to reserve mutex X, task A attempts to reserve mutex Y, the two tasks are mutually waiting for each other). The mutex deadlock prevention feature is based on a reservation *rank*. The mutexes must be reserved in increasing order of their rank (member **`deadlock_prevention_rank`**). There is no

restriction on the order of mutex relinquishing operations. For the software test planning, this feature provides repeatable error notification (symptomatic of an embedded software design flaw) whenever the normal operation of the software is tried. Without the deadlock prevention feature, the design flaw would cause intermittent mutex deadlock, which is usually much harder to diagnose.

3.4.4 Implementation Issues

The performance characteristics of mutual exclusion semaphores are quite good: in the usual case where the **`mutex_reserve`** operation finds the mutex available, the kernel scheduler is not involved, and no context switch occurs. In the other case, the kernel scheduler performs an essential task for resource access serialization. In other words, the mutex mechanism performance penalty occurs precisely when the system coherency would be jeopardized if the synchronization requirement had been overlooked.

The mutex implementation uses a double linked list for keeping track of mutexes currently owned by each task. This list is sorted by member **`deadlock_prevention_rank`**. Also, each mutex is the head of a simply linked list of tasks currently waiting for the mutex.

3.5 'D' - A Queuing Mechanism

3.5.1 Overview

3.5.1.1 A Simple Memory-Based FIFO Mechanism

The ABCD Proto-Kernel™ queuing mechanism implements a memory-based FIFO scheme with fixed allocation of memory space. The term "event queue" refers to this queuing mechanism, although it is somehow an abuse of language since the ABCD Proto-Kernel™ queues are not specific to "events."

The basic event queue construct is fairly straightforward. Basically, the event queues are the simple programming construct that a skilled programmer would design from a memory-based FIFO requirement:

- a **`next_in`** indicator (designating the empty entry where the next insert operation will occur) and a **`next_out`** indicator (designating the occupied entry that will be processed by the next extract operation), with wrap-around arithmetic,
- queue empty condition exists when **`next_in == next_out`**,
- a queue full condition exists if advancing the **`next_in`** indicator to insert a new entry would cause **`next_in == next_out`** with the FIFO is fully occupied instead of empty.

By itself, such a FIFO mechanism requires no explicit synchronization between the entry

inserting task and the entry extracting task.

Note: This FIFO mechanism can be used between independent processors sharing a common memory. Indeed, it was used in the 1970's in Control Data scientific computers (Cyber CDC6000) with the FORTRAN library disk I/O between the main CPU and co-procesors then known as “peripheral processor units.”

There is a small number of event queue functions. Notably, there is no current entry count maintained by the event queue interface. Also, there is no facility to look at the entries in the queue over the oldest one (a “queue peek” function).

Note: The rationale for this lean interface to event queues is the simplicity that an embedded system software should exhibit. The software components that insert queue entries should not be concerned with the pace at which the entries are retrieved. If an application requirements calls for some notion of queue entry priorities, the recommended implementation uses an independent queue for each priority class and a combined queue for signaling the extracting task. In this example, the application complexity is handled by repetitive use of a simple kernel mechanism.

3.5.1.2 Embedded Software Refinements

The ABCD Proto-Kernel™ event queue mechanism is harmonized with other kernel concepts. Moreover, some refinements are made to the simple memory-based FIFO mechanism to adapt them to the field of embedded software. These refinements are detailed in the following sub-sections. Taking an overview perspective, a brief description is given below:

3.5.1.2.1 Task Blocking on One or Two Queues

The preferred application task structure is a polling loop that processes events retrieved by the task from one or two queues. This capability is supported with the functions the **`abcd_wait_one_queue`** and **`abcd_wait_two_queues`**.

3.5.1.2.2 Support of Event Insertion by Interrupt Service Routines

An ISR is allowed to insert entries in the event queues relevant to the device that they service. This is indeed the preferred way of signaling from an ISR to an application task. It is supported both in ISRs written in C/C++ and in assembler. In the latter case, programmer-friendly macros expand as optimized instruction sequences.

3.5.1.2.3 Support of Multiple and Potentially Overlapping Insertion Operations

The event queue mechanism allows multiple tasks to insert entries in a queue, with built-in synchronization. If a task A begins an entry insertion and then task B begins and

complete an entry insertion in the same queue, the B's entry will not be seen by a task doing the queue entry extraction until the A's entry has been completed.

3.5.1.2.4 Preventive Programming Support for Queue Full Conditions

The queuing mechanism has special provisions for the queue full condition. With these provisions, the application software need not systematically check the validity of the entry pointer used for queue entry insertion. If a queue full condition arises, the queue entry data will be lost, but the software integrity will otherwise remain intact. In many cases with actual real-time applications, this corresponds the desired behavior because the system should ignore untimely or delayed data.

3.5.1.2.5 Event Queues Devoid of Queue Data Memory

An event queue can exist without queue entry space. This represents a space-efficient means of signaling from either an ISR or tasks to a task.

3.5.1.3 ABCD Event Queue Usage Sample Code

In this subsection, sample source code is shown using an *equivalent class declaration* for a typical queue. The equivalent class declaration is used in order to introduce the run-time queue concepts without the class derivation and template details.

The C++ template mechanism is used for ABCD Proto-Kernel™ queues to affix an entry data type to an event queue, and to enforce type checking for management of queue entries by application programs. The application programmer shouldn't be intimidated by the C++ language features used in the ABCD Proto-Kernel™ programming interface: they do not force the application software to use any of the most advanced C++ features. The following sample code shows the effect of applying the class derivation and template mechanisms.

3.5.1.3.1 Type and Object Declarations and Definitions

The sample queue entry is a data structure defined as

```
struct event_entry_str
{
     unsigned long event_code;
     unsigned long event_data[2];
};
```

The memory reserved for the queue entries would look like:

```
#define EVENT_ENTRY_COUNT (40+2)
struct event_entry_str queue_data[EVENT_ENTRY_COUNT];
```

The *proper* declaration of the event queue would look like:

```
class abcd_ev_queue<struct event_entry_str> sample_queue
                              (queue_data,EVENT_ENTRY_COUNT);
```

The proper code fragment **class abcd_ev_queue<struct event_entry_str>** has an equivalent class declaration that looks like this:

```
class abcd_ev_queue_equivalent_class
{
protected:
   // Queue object data members
   long entry_size; // Size of each entry in the queue data
   long last_disp;  // Highest displacement for wrap-around
                    // arithmetic
   long next_in;    // The IN indicator in the FIFO logic
   long next_out;   // The OUT indicator in the FIFO logic

   long in_front;   // The advanced IN indicator
   unsigned long front_cnt;   // The number of outstanding
                              // advanced IN objects

   struct event_entry_str * ptr_base;   // Pointer to the queue
                                        // data area

public:
   // Queue object constructor
   abcd_ev_queue_equivalent_class
               (struct event_entry_str *ptr_bas, int ent_cnt);

   // Queue insertion operation in an ISR context
   volatile struct event_entry_str * get_in_entry_isr();
```

```
    // Queue insertion operation in a task context
    volatile struct event_entry_str * get_in_entry();
    void advance_in(volatile struct event_entry_str *entry);

    // Informative function to detect a queue full condition
    volatile struct event_entry_str * get_queue_full_entry();

    // Current queue status for the extraction operation
    int is_empty();

    // Queue extraction operation
    struct event_entry_str * get_out_entry();
    void advance_out();
};
```

Given this equivalent class declaration, the queue object is defined by:

```
class abcd_ev_queue_equivalent_class sample_queue
                              (queue_data,EVENT_ENTRY_COUNT);
```

This declaration implicitly calls the queue constructor function, that initializes the queue object data members (details are irrelevant for the application programmer), and logically assigns the queue_data array to the queue.

External linkage (use of the above objects **queue_data** and **sample_queue** by another source file) is supported with the following external object declarations:

```
extern struct event_entry_str queue_data[];
extern class abcd_ev_queue_equivalent_class sample_queue;
```

From this point on in the explanation of event queues implementation, the sample source code is not impacted by the above use of an equivalent class declaration.

3.5.1.3.2 Queue Insertion Operation in a Task Context

The queue insertion operation in a task context is done with the function pair **get_in_entry()** and **advance_in()**. Together, these two calls ensure proper serialization of queue entry insertions in the ABCD Proto-Kernel™ task execution environment.

```
{
   // Queue insertion in a task context
   volatile struct event_entry_str * new_entry;

   new_entry=sample_queue.get_in_entry();
   new_entry->event_code=0xBACABA; // or so
   new_entry->event_data[0]=0x1; // or so
   new_entry->event_data[1]=some_function_with_side_effect();
   sample_queue.advance_in(new_entry);
   // Further uses of the object *new_entry are disallowed
}
```

The above code sequence works even if the queue is full, in which case the pointer **`new_entry`** will have a value equal to **`sample_queue.get_queue_full_entry()`** or **`queue_data`**. Unless a queue full condition exists, the function pair **`get_in_entry()`** and **`advance_in()`** must be completed to prevent the queue from being locked (function pairs **`get_in_entry()`** and **`advance_in()`** later performed by other tasks would remain ineffective). Since there is no precise deadline for the call to **`advance_in()`**, the ABCD Proto-Kernel™ has no means to enforce that it is eventually called.

The queue full condition may be handled explicitly by an application code sequence. If the application requirements suggests that the function **`some_function_with_side_effect()`** should not be called when the queue is full (e.g. the queue full condition is symptomatic of a system overload condition and the function call would only worsen the situation), the code sequence below can be used:

```
{
   // Queue insertion with queue full provision
   volatile struct event_entry_str * new_entry;

   new_entry=sample_queue.get_in_entry();
   if (new_entry!=sample_queue.get_queue_full_entry())
   {
      new_entry->event_code=0xBACABA; // or so
      new_entry->event_data[0]=0x1; // or so
      new_entry->event_data[1]=some_function_with_side_effect();
      sample_queue.advance_in(new_entry);
   }
```

```
}
```

3.5.1.3.3 Queue Insertion Operation in an ISR Written in C or C++

An ISR written in C or C++ can use a single queue function for the insertion operation (it may also use the same function pair as a task).

```
void isr_sample( struct abcd_isr_table_entry_str *isr_entry )
{     // Queue insertion in an ISR context
      volatile struct event_entry_str * new_entry;

      new_entry=sample_queue.get_in_entry_isr();
      new_entry->event_code=0xDADA; // or so
      new_entry->event_data[0]=0x2; // or so
}
```

Just like in a task context, the queue full conditions can be ignored or processed as the circumstances may require.

3.5.1.3.4 Queue Insertion Operation in an ISR Written in Assembler

See section 6.2.2.1 on page 66 for explanations on the following code sample.

```
      .text
//    extern void sample_isr(void);
//
      .globl    _Z10sample_isrv
_Z10sample_isrv:    // Nothing to do

      ABCD_SAVE_REGS(2,0)
      lis  %r24,sample_queue@ha
      la   %r24,sample_queue@l(%r24)
      QUEUE_GET_IN_PTR(1254)
      li   %r26,0xBED
      stw  %r26,0(%r25)         // new_entry->event_code=0xBED;
      li   %r26,0x3
      stw  %r26,4(%r25)         //new_entry->event_data[0]=0x3;
```

```
ABCD_ISR_EPILOGUE(SHRT_MAX,2,1254)

DECLARE_QUEUE_EXCEPTION_PROCESSING
.previous
```

3.5.1.3.5 Queue Extraction Operation

The queue extraction operation is supported in C or C++ only. It uses the function pair **`get_out_entry()`** and **`advance_out()`**. The queue extraction is often synchronized with the ABCD Proto-Kernel™ scheduling functions **`abcd_wait_one_queue()`** or **`abcd_wait_two_queues()`**, but this is not always so, as shown in the example below:

```
{
   // Queue extraction
   while (!sample_queue.is_empty())
   {
      struct event_entry_str * retrieved_entry;
      retrieved_entry=sample_queue.get_out_entry();
      switch (retrieved_entry->event_code)
      {
         case 0xBACABA:
            // process an event from section 3.5.1.3.2 of
            // this document
            break;
         case 0xDADA:
            // process an event from section 3.5.1.3.3 of
            // this document
            sample_queue.advance_out();
            // further processing that can not
            // refer to object *retrieved_entry
            retrieved_entry=NULL;
            break;
      }
      if (NULL!=retrieved_entry)
         sample_queue.advance_out();
   }
}
```

The call to `sample_queue.get_out_entry()` provides a pointer to the next event (queue entry) to process, and its source is given by `retrieved_entry->event_code`. The call to `sample_queue.advance_out()` releases the queue entry memory, so that a) it becomes available for other coming event insertions, and b) the call to the function `sample_queue.is_empty()` eventually returns non-zero. If the processing of event code `0xDADA` is relatively long, the early call to `sample_queue.advance_out()` can slightly reduce the probability of a queue full condition for the coming event insertions.

Except for the fact that it is not synchronized by a call to the function `abcd_wait_one_queue()` or `abcd_wait_two_queues()`, the above code sample shows the typical use of an task event queue in a ABCD Proto-Kernel™ application software. While this code sequence runs, interrupt signals can trigger ISR execution and the ABCD Proto-Kernel™ scheduling may suspend this code sequence execution to let higher priority tasks use the CPU. This is an inherent characteristic of a task execution that it can generally be quietly suspended by ISRs and task scheduling. Due to this possible execution suspension, queue entries may be inserted in the queue `sample_queue` while entries are retrieved in the above code example.

3.5.2 Event Queue Declaration and Initialization

The ABCD Proto-Kernel™ queues are C++ class objects. There is one base class, `class abcd_ev_queue_base_cl`, and three *derived* classes, respectively

- `template<class T> class abcd_ev_queue`, for normal queues with an associated entry type `T`, the actual queue objects are *template instantiations* following the C++ terminology and source software syntax,
- `class abcd_ev_queue_cl`, for normal queues where the entry type is not enforced by the compiler (can be attractive to developers having no faith in the C++ template mechanism),
- `class abcd_index_queue_cl`, for queues devoid of entry memory.

It is not possible to create an object of type base `class abcd_ev_queue_base_cl`. The address of any of the derived and instantiated class object above can be used as a pointer to the base `class abcd_ev_queue_base_cl`, especially useful for the task scheduling functions `abcd_wait_one_queue()` or `abcd_wait_two_queues()`.

This class derivation and template instantiation is shown in figure 2.

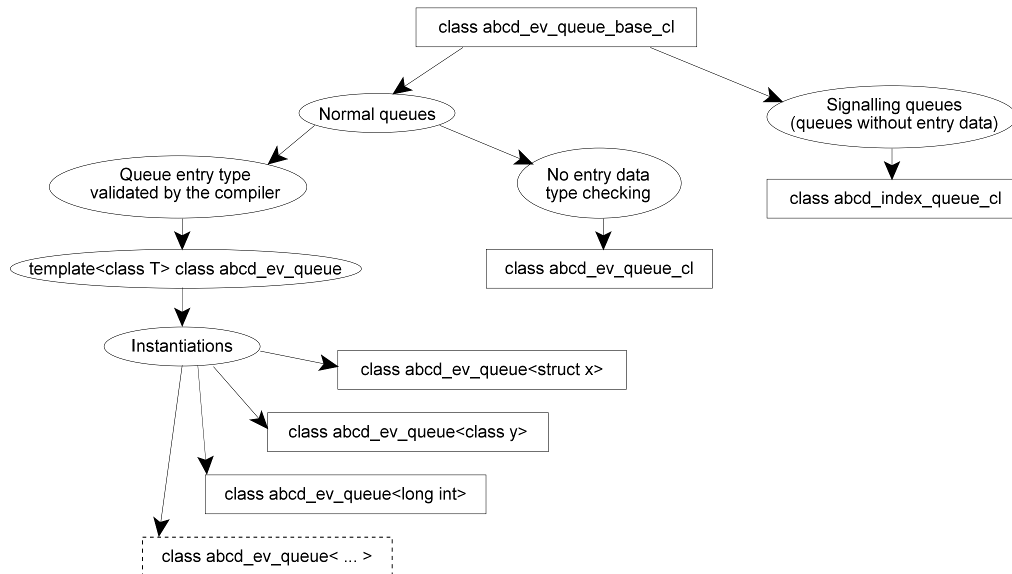

Figure 2. ABCD Proto-Kernel™ Queue Classes Hierarchy

3.5.2.1 Usual Event Queue Declarations and Definitions

The event queue entries are allocated as an array of fixed size memory blocks. The array address is passed as the first argument to the queue class constructors. The queue entry size and the queue entry count in the array is fixed and recorded in the queue object through the individual queue constructor. In the case of queues declared with the template instantiation mechanism, the size of a queue entry is implicit from the queue entry type. The typical usage of queue constructors in an embedded application software is statically allocated queue objects, of global scope, with fixed memory reservation for the queue entry data.

```
class abcd_ev_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // constructor
```

```
     abcd_ev_queue_cl
          ( void *ptr_bas     /* event queue entries, an array of
                                 fixed size memory blocks */
          , int ent_cnt       /* queue entry array count */
          , int ent_siz);     /* individual queue entry size */
//...
};

template<class T> class abcd_ev_queue
                              : public abcd_ev_queue_base_cl
{
//...
public: // constructor
     abcd_ev_queue
          ( T *ptr_bas        /* event queue entries, an array of
                                 T objects */
          , int ent_cnt);     /* queue entry array count */
//...
};
```

The first entry in the array is the special one used by the queue implementation for the queue full condition.

The maximum concurrent queue entry count (before a queue full condition occurs) is 2 fewer than the array entry count. It is an application software design parameter that can be set conservatively high, and later adjusted if needed, e.g at system integration time when the impact of a queue full condition might be observed.

Typically, an event queue entry holds a structure with an event code as its first element, followed by event-code-specific detail data. The ABCD Proto-Kernel™ itself does not impose this entry arrangement, but the generic timer facility (see section 5.2.2 on page 48) relies on the presence of a 32 bits code at the beginning of event queue entries to facilitate the merging of generic time-out events in application-specific queues.

3.5.2.2 Event Queue Definitions for Queues Devoid of Entry Memory

An event queue devoid of queue data memory is created from the type `class abcd_index_queue_cl` using either the default constructor, as in

```
          class abcd_index_queue_cl y;
```

for a queue **y** with a maximum of INT_MAX**-2** insertion operations before a queue full condition arises, or the constructor with an **int** parameter that specifies the value queue entry count plus 2, as in

```
        class abcd_index_queue_cl x(102);
```

for a queue **x** with a maximum of 100 insertion operations before a queue full condition arises.

```
class abcd_index_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // constructors

     abcd_index_queue_cl();

     abcd_index_queue_cl(int ent_cnt);

//...
};
```

3.5.3 Queue Entry Insertion Functions

```
class abcd_index_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // Queue insertion fonctions
     int  get_queue_full_entry();

     int  get_in_entry_isr();

     int  get_in_entry();
     void advance_in(int entry);

//...
};

class abcd_ev_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // Queue insertion fonctions
```

```
     void * get_queue_full_entry();

     void * get_in_entry_isr();

     void * get_in_entry();
     void advance_in(void *entry);

//...
};

template<class T> class abcd_ev_queue
                              : public abcd_ev_queue_base_cl
{
//...
public: // Queue insertion fonctions
     volatile T * get_queue_full_entry();

     volatile T * get_in_entry_isr();

     volatile T * get_in_entry();
     void advance_in(volatile T *entry);

//...
};
```

3.5.3.1 Queue Entry Insertion by Tasks

The queue entry insertion is achieved by the task by the following three step procedure:

- a call to the queue class function **`get_in_entry()`** that returns a queue entry pointer,
- setting data values in the queue entry indicated by this pointer,

  Note: In data-intensive embedded applications, copying memory buffers should be avoided. Thus, the queue entry data values should usually be pointers to data buffers rather than the buffer contents.
- a call to the queue class function **`advance_in()`** with the queue entry pointer passed as an argument.

This three step procedure allows multiple concurrent insertion in the queue (a built-in synchronization property of the ABCD Proto-Kernel™ queues). If a task A begins an entry

insertion and then task B begins and complete an entry insertion in the same queue, the B's entry will not be seen by a task doing the queue entry extraction until the A's entry has been completed. Moreover, this three step procedure transparently supports the queue full condition (see section on page 3.5.3.4 on page 41)

3.5.3.2 Queue Entry Insertion by Interrupt Service Routines

The queue class function **`get_in_entry_isr()`** (taking no argument) does for ISRs coded in C++ what the function **`get_in_entry()`** does for tasks, except that ISRs need not call any function comparable with the **`advance_in()`** function. The queue entry insertion is implicitly completed when the ISR finishes its execution.

The ABCD Proto-Kernel™ allows queue entry insertion by ISRs written in assembly language. The same processing principles apply to both types of ISRs. This is detailed in section 6.2.2 on page 66 for the Motorola MPC8xx processor family.

3.5.3.3 Explicit Kernel Scheduling Invocation After Entry Insertion

Let's suppose the high priority task A waits on its event queue. When a lower priority task inserts an entry in an A's event queue, the kernel scheduler is not automatically invoked as if the entry insertion has been done by an ISR. The lower priority task should explicitly call the function **`abcd_scheduler_run`** (taking no argument) to achieve a system behavior coherent with the prioritization of tasks.

Note: This is the only circumstance in which an explicit application action is needed to enforce the scheduler operation in compliance with its specifications (allocating the CPU to the highest priority task that is not blocked). With the mutexes, a relinquish operation quietly invokes the scheduler, when appropriate. The scheduler is also quietly invoked when interrupt sources are enabled, in a controlled way in anticipation of multiple ISRs being triggered by a single interrupt enabling operation.

3.5.3.4 Queue Full Condition

A queue full condition is reported by the queue class member function **`is_empty()`** (taking no argument). However, this function is an instantaneous test that might be wrong at any moment after being made (e.g. if the queue is nearly full when the test is made and filled by another task immediately after the test).

Two better methods exist to handle the queue full condition. If an application (task or ISR) attempts to insert an entry in a queue that is full, it receives an entry pointer where the queue data can actually be written without harm to the remaining of the system, instead of a null pointer that would indicate a queue full condition. With this kernel behavior, the application software

designer has the option to systematically assume the validity of the entry pointer used for queue entry insertion (provided that inserted data loss is acceptable). The alternate application processing strategy is to test the entry pointer against the queue class member function **`get_queue_full_entry()`** as in the following code fragment:

```
new_entry=sample_queue.get_in_entry();
if (new_entry!=sample_queue.get_queue_full_entry()) {
   /* ... */
   sample_queue.advance_in(new_entry);
}
```

As a subtle consequence of the kernel behavior for the queue full condition, the entry pointer used for queue entry insertion should be considered **`volatile`**: whenever two program sequences (ISR or task) attempt to insert entries concurrently while the queue is full, they write to the same entry memory area without any synchronization provision.

Note: It might be legitimate for a "glue" function that returns an entry pointer for entry insertion to deliberately return the pointer reserved for queue full condition as a substitute for the **`NULL`** pointer value, e.g. when the function detected a related lack of resource for a valid entry insertion.

3.5.4 Queue Entry Extraction

```
class abcd_index_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // Queue extraction functions
     int is_empty();

     int  get_out_entry();
     void advance_out();
//...
};

class abcd_ev_queue_cl : public abcd_ev_queue_base_cl
{
//...
public: // Queue extraction functions
     int is_empty();
```

```
     void * get_out_entry();
     void advance_out();
//...
};

template<class T> class abcd_ev_queue
                              : public abcd_ev_queue_base_cl
{
//...
public: // Queue extraction functions
     int is_empty();

     T * get_out_entry();
     void advance_out();
//...
};
```

3.5.4.1 Queue Entry Extraction Functions

There is no synchronization feature for the event queue extraction functions. A single task should be assigned the responsibility to extract the entries from an event queue. If an application requires to do otherwise, other synchronization mechanism (e.g. a mutex) should be used to prevent two concurrent queue extraction operations.

3.5.4.2 Queue Empty Condition

The queue empty condition can be tested with the queue class function **`is_empty()`** (taking no argument).

3.5.4.3 Event Queue Wait Functions

The ABCD Proto-Kernel™ blocking functions **`abcd_wait_one_queue()`** and **`abcd_wait_two_queues()`** provide the tasks with the capability to block itself waiting for event to be inserted in one or two queues. The queue indications are passed as one or two queue class pointer arguments to these function calls.

According to the C++ class derivation rules, the kernel functions **`abcd_wait_one_queue()`** and **`abcd_wait_two_queues()`** are generic and applicable to any type of ABCD Proto-Kernel™ queue object.

The kernel does not enforce that a single task waits for a given queue. It is up to the application to apply such a rule as it usually applies.

4. Strategies for Mission-Critical Interrupt Latency

4.1 Support for Delicate Balance of Priorities for to Achieve Adequate ISR Latencies

4.1.1 Introduction

We do not state a definition of "ISR latency" that would allow us to tell whether the "adequate ISR latency" has been achieved in a given application context. The ISR latency broadly refers to the delay between the external event that triggers an ISR to the required software reaction. In the ABCD Proto-Kernel™, a software reaction is either within the ISR itself or in a task which is awakened by the ISR.

In a designer's search for adequate ISR latency, one must keep in mind the worst-case latency. The worst case latency is the longest latency possibly experienced by a system when measured between a specific external event and a specific software reaction. It usually occurs when a cascade of external events are unexpectedly well synchronized. While we can't tell what an adequate latency is, we can identify specific delays and processing times that contribute to the worst case latency.

The interrupt latency analysis assumes a worst case scenario for the variable delay between an external event and the software action by a task. In a real-time system, another limit condition exists if the external event occurs at an excessive frequency. In this case, the software action by the task would be delayed because the task is still busy processing a preceding external event occurrence. The following discussion assumes that the system meets this boundary condition on the external event frequency.

When an embedded system experiences operating problems intermittently, a possible cause is an inadequate worst-case latency (of course another possible cause is the lack of a required synchronization mechanism). With the processing power available in a PowerPC processor and the presence of the timebase high resolution timer, it is possible to monitor the latency in the very software that may be affected if it is too long. Don't expect these measurements to be perfect. In the laboratory, an oscilloscope can do marvels at revealing the true latency of something.

4.1.2 The First Approximation Requirement for Nesting of ISRs

Suppose we have a class of villain ISRs that take up too much time to complete, and a class of critical interrupts that complete quickly but can not wait for any in-service villain ISR to complete. Assume for a moment that the ABCD Proto-Kernel™ was implementing nested ISRs. The villain ISR would enable other interrupts to be serviced during its own execution. This would create an execution context of a higher priority than the highest priority task, with questionable benefits, except in the rare occasion where the high priority ISR processing is self-complete (i.e. the adequate software latency is not dependent on a task) *and* the villain ISR processing can not be deferred to a task. Without recourse to any subtle ABCD Proto-Kernel™ concepts described later, there is a simple alternative to nested interrupts: perform the bulk of the critical interrupt processing in a higher priority task, and move the bulk of the vilain ISR processing to a lower priority task.

4.1.3 Better than Nested ISRs, a Critical Real-Time Arena

Suppose we face a genuinely demanding interrupt processing latency requirement. Typically, a time-critical process would be handled by a very small set of interrupt sources and an small set of high priority tasks. Other processes would be handled in parallel by the embedded system, but with much less stringent time-criticality. As far as the interrupt dispatching function is concerned, the first information to look for is the relative priorities assigned to the various interrupt sources by the interrupt controller logic. Then, the worst-case ISR latency has two potential contributors:

a) From the high priority interrupt sources, a potentially burdensome latency comes from the statistically rare near-simultaneous occurring of multiple high priority events.

b) Low priority villain ISRs may remain a problem if they can not be coerced into well-behaved ISRs (i.e. the bulk of their processing deferred to a task). The longest such low priority villain ISR defines the worst case latency from low priority sources.

We thus define the *critical real-time arena* as the set of ISRs that are bound to a time-critical process and of higher priority than the *arena's ISR priority threshold* plus the small set of tasks of higher priority than the *arena's task priority threshold*. With the critical real-time arena, the ABCD Proto-Kernel™ associates a set of *arena-prevented ISRs* which are deliberately masked when the CPU runs in a context within the critical real-time arena. ISRs of higher priority, and perhaps of lower priority if they are villain, can be liberally be allocated to the set or arena-prevented ISRs. Some innocuous high priority ISRs may be left out of the set, e.g. for alarm conditions.

The ABCD Proto-Kernel™ services to ISRs allow them to determine that kernel scheduling is not needed upon exit, based on the current task's running priority. In order

to reduce the worst-case latency experienced by tasks within the critical real-time arena (specifically reduced by the processing time that the ABCD scheduler would take to re-select the very same interrupted task), this feature should be used by ISRs outside of the critical real-time arena and arena-prevented ISRs.

In the detailed design of an embedded application where the arena may be needed, the following considerations should be taken into account:

a) The PowerPC decrementer interrupt function is better assigned to something categorized as an innocuous low priority ISR (because it is non-maskable and of lowest priority among all interrupt sources).
b) For the grouped masking interrupt sources, the designer loses the convenience of precise definition of the set of arena-prevented ISRs.
c) The ABCD Proto-Kernel™ implementation does not allow for more than one level of nested critical real-time arenas. There are no anticipated real life uses for more than a single critical real-time arena in an embedded system.
d) Sometimes a single interrupt source combines the received data signal from multiple input channels, leading to a situation where an ISR services an unknown number of devices before completion. This may qualify the ISR as a villain in the present discussion, but otherwise the situation is easily handled by the ABCD Proto-Kernel™ facilities, notably the queuing mechanism. The design recommendation is to assign the multi-channel processing to a task with the priority just below the critical arena threshold.
e) A bouncing switch or relay is likely to trigger a large number of interrupts in a short period in the order of 2 to 10 milliseconds. These are not necessarily candidates for the critical real-time arena. For alarm indications, it is sometimes adequate to disable the interrupt source in the ISR itself, and then re-enable it in a task after some delay. The de-bouncing of a manually operated switch (recognition that the switch has been in a stable position for a duration of a fraction of a second), when done by the software, is usually done with a polling strategy, despite the relatively high CPU burden for a seemingly trivial task.

The application of the ABCD real-time critical arena alters the interrupt signal responsiveness in subtle ways, and should be used with an in-depth understanding of the system hardware and software. Example:

> High priority task A reserves a mutex currently owned by low priority task C. Intermediate priority task B is ready to run, and within the critical real-time arena. If C is waiting for an interrupt signal masked while B runs, the task A becomes delayed by application of the real-time arena mechanism. This is an unlikely design since it is not recommended to let a task block itself while it owns a mutex.

4.1.4 Conclusion

Overall, the critical real-time arena scheme is deemed sufficient for most applications. The key point is to restrict the amount of processing which is done inside the critical real-time arena so that the worst case latency for other processing is acceptable. If an application can not fit this model, it is symptomatic of insufficient or badly allocated hardware processing power. Nowadays, truly time-critical processes tend to be supported by programmable logic, FIFO memories, dual-port memories, DMAs, dedicated processors, coprocessors like the MPC8xx CP, and the like, instead of by a general purpose CPU. Moreover, the modern hardware designs usually contain logic to detect excessive latencies (receiver overruns, transmitter underruns, alarms triggered by late acknowledgment of ready signals). The ABCD Proto-Kernel™ critical real-time arena concept is meant to accommodate demanding hardware designs, if any still exists.

In a prototype hardware design, a requirement for a hardware solution to a potential latency bottleneck is sometimes overlooked. The ABCD Proto-Kernel™ dedication to hard real-time requirements offers the potential to deliver a working prototype in these cases.

5. Useful Additions to the ABCD Proto-Kernel™

5.1 Buddy System Memory Allocation

A simple "buddy system" dynamic memory allocator is included in the ABCD Proto-Kernel™ in order to facilitate buffer allocation for deeply embedded systems in which the execution latency is critical. The typical modern dynamic memory allocators are tuned for run-time environments having memory paging capabilities, and are not easily adapted to the deeply embedded systems environment. The ABCD Proto-Kernel™ buddy system dynamic memory allocator is provided as a general purpose mechanism for embedded system environment. The name "buddy system" refers to the internal organization of the memory allocator, a subject matter covered by textbooks on operating system principles.

When designing an embedded application software, the need for dynamic memory allocation can turn into specialized memory allocators (e.g. separate receive buffer pools for every communications ports). The alternative strategy is to rely on a single general purpose memory allocator. The specialized allocator strategy makes the software design tuning more tedious, but may exhibit unique run-time reliability characteristics, i.e. a lack of buffers for a communications port does not block the other ports, or other system functions.

With the ABCD Proto-Kernel™, it is sometimes useful for a task 'A' to allocate a memory block from dynamic memory, then pass a pointer to this block to another task 'B' that will eventually

release the block. Because the dynamic memory allocation facility is inherently not reentrant, not all implementations of the C standard functions **malloc** and **free** support this. The ABCD Proto-Kernel™ buddy system memory allocator implements the required mutual exclusion semaphores so that tasks can allocate (**malloc** or related functions) and release (**free**) dynamic memory without paying any attention to "memory ownership."

The buddy system memory allocators provide memory blocks in a small number of predefined sizes (usually a power of two times the smallest block size). Thus, the actual memory size actually used is often larger than the requested size. The ABCD Proto-Kernel™ buddy system allocator features the **allocsize** function that returns the actual block size for a dynamic memory block previously allocated. This function is declared as:

```
extern size_t allocsize(void *r);
```

This function can be used by applications to optimize the number of allocated memory blocks when using multiple buffers in a segmentation scheme. E.g. where the requested size was 768 bytes (argument to the **malloc** function) a 3KB protocol packet can fit in three 1024 bytes buffers based on the actual buffer size (returned by the **allocsize** function).

5.2 Timer Facilities

5.2.1 Standard Type clock_t Support

The implementation of the standard type **clock_t** is controlled with the preprocessor symbol ABCD_CLOCK_T_SOURCE that can take two possible values:

> ABCD_CLOCK_T_SOURCE_TIMEBASE, when the standard type **clock_t** is based on the MPC8xx timebase facility, or
>
> ABCD_CLOCK_T_SOURCE_RTC, when the standard type **clock_t** is based on the MPC8xx real-time clock.

5.2.2 General-Purpose Timers

No tick timer (i.e. of the kind used for round robin scheduling) is provided among the useful additions to the ABCD Proto-Kernel™. An embedded system design should use the hardware timers in ways that suit the application requirements.

A generic timer facility is provided. The timer periods are expressed in units of millisecond. However, the timer resolution may be coarser than one millisecond.

The general-purpose timer functions are declared in the file abcd_utils_wait_fncs.h.

For the implementation on the Motorola MPC8xx Processor Family, the use and implementation of general purpose timers is controlled with the preprocessor symbol ABCD_WAIT_TIMER_SELECT that can take three possible values:

ABCD_WAIT_TIMER_NONE, when the general purpose timers are not needed,

ABCD_WAIT_TIMER_RISCT, when the general purpose timers are to be based on the MPC8xx RISC timers table, or

ABCD_WAIT_TIMER_TMR_TICK, when the general purpose timers are to be based on the system tick timer (see section 5.2.3 on page 50).

With the ABCD_WAIT_TIMER_TMR_TICK alternative, the timer table processing is implemented in a dedicated kernel task, typically of low priority. This design is fine if the application can live with timer jitter related to the instantaneous processing load in other application tasks. Other implementations are possible, notably the timer table processing implemented in the ISR itself.

5.2.2.1 Blocking Wait Function

A blocking wait function is provided with:

```
int abcd_util_wait(unsigned long period_millisec);
```

This function is blocking and generally should be used with caution, especially in high priority tasks where worse-case latency for external signal processing is usually critical. The return value is non-zero on error.

5.2.2.2 Wait Functions for Queued Timer Expiry Events

A more flexible wait function is provided to insert a timer expiry event code in a task's event queue with:

```
int abcd_util_wait( unsigned long period_millisec,
                    class abcd_ev_queue_cl *queue_a,
                    int event_code);
```

In the above function call, the queue pointed by the **queue_a** parameter must have an entry array (and not a queue devoid of queue data memory). Moreover, the queue entry structure must be such that the first 32 bits are the suitable memory location to store the **event_code** value, in

order for the task to recognize the timer expiry event among other entries. This function return value is less than zero on error, and a *timer index* otherwise. This timer index is used in the next two function calls for timer management.

Once a timer expiry event is requested by this last function, it may be cancelled with the function:

```
int abcd_util_cancel_wait(int i_timer);
```

This timer cancellation function must be called no more than once for each call to **abcd_util_wait**. If the cancellation is properly completed, the return value is zero. On error, the return value is less than zero. If the cancellation request came too late to be effected, the return value is greater than zero, and the application software should behave as if it didn't request this timer cancellation.

After the timer expires, the application software will detect the timer expiry event in a task's completion queue as indicated by the parameters to the call to **abcd_util_wait**. A call to the function:

```
void abcd_util_ack_wait(int i_timer);
```

is then requested.

5.2.3 System Tick Timer

The term *system tick timer* usually refers to a the time-slice timer in a round-robin kernel, where tasks in a given priority class share the CPU in time-slices. The ABCD Proto-Kernel is *not* a round-robin kernel. Its system tick timer is a useful addition. The system tick timer is a mere programming convention for a periodic timer to which application-specific polling activities can be attached.

Note: The need for a system tick timer occurs when using both 1) general purpose timers (section 5.2.2 on page 48), and 2) idle CPU power saving (section 6.6.1 on page 75), but only if the doze low mode is selected for power saving. This is because the RISC timers are counting at a fraction of the expected rate when the low clock divisor is used. Obviously, the need for a system tick timer can occur from a number of application software requirements.

The use and implementation of the system tick timer is controlled with the preprocessor symbol ABCD_TIMER_TICK_SELECT that can take three possible values:

ABCD_TIMER_TICK_NONE, when the system tick timer is not used,

ABCD_TIMER_TICK_DECREMENTER, when the system tick timer is to be based on the MPC8xx decrementer,

ABCD_TIMER_TICK_PIT, when the system tick timer is to be based on the MPC8xx programmable interval timer.

5.3 The FlashCnl Software Library

The FlashCnL software library is included as an ABCD Proto-Kernel™ useful addition. This library is described in the reference [FLASHCNL]. This library provides system configuration data management facilities and log recording facilities.

5.4 Embedded Interactive Monitor Facilities

The trace and the maintenance command mechanisms described in the two following document subsections provide a basic interactive system monitor function. As a useful addition to the ABCD Proto-Kernel™, neither of these is required in any embedded system. If used, the application design plays a very important role in defining their purpose, scope and detailed specifications.

In their simplest implementation, trace and the maintenance command mechanisms support a serial port of the type found in the Motorola MPC8xx processor family using a subset of Internet RFC 1549 as framing specifications (asynchronous HDLC protocol, PPP framing, see reference [RFC1549]). The same protocol is easily supported by any processor architecture supporting an asynchronous serial port. This complies to the ABCD Proto-Kernel networking specifications that are described in the document [LAB_COMM_UG]. This same document also covers a software utility, lab-comm, that is compatible with the ABCD Proto-Kernel™ serial port traces.

Note: The Motorola MPC8xx processor family and the PPCMB/850 processor board offered by CONNOTECH has a great potential for networking connectivity. For now, a reasonable coverage of this subject area for a novice reader is outside the scope of this document.

5.4.1 Traces Through a Communications Channel

The function **tprintf** uses the same calling conventions as the C standard function **printf**, except that it transmits the output string through a communications channel assigned to the serial port traces facility. This creates a situation where a valuable resource (a communications channel) is assigned to a useful addition to the ABCD Proto-Kernel™, with potential conflicts with the application requirements.

Currently, the function **`tprintf`** transmits a *frame payload* that is *encapsulated* in a *layer 2 protocol frame* according to the ABCD Proto-Kernel™ networking specifications (see document [LAB_COMM_UG]).

5.4.2 Maintenance Commands

The ABCD Proto-Kernel™ maintenance commands is a useful addition available to application software developers for commands typically used by service personnel for system configuration, test, and diagnostics. The receipt of command lines through one or more communications channel is either left as an application design issue, or follows the ABCD Proto-Kernel™ networking specifications (see document [LAB_COMM_UG]). The present document subsection gives a high level description of the parsing of maintenance commands and the initiation of application-specific execution of validated commands. The source files should be used as a detailed reference. See the files abcd_maintcmd.cpp and abcd_maintcmd.h. The files abcd_maintcmd_dispatch.cpp and abcd_maintcmd_dispatch.h contain application-specific definitions, table initializations and switch/case statements that implement the application usage of the maintenance commands useful addition.

Once a command line is passed to the maintenance command facility, the generic command-line parsing occurs, followed by a table-driven dispatching of commands. The design goal is to make is easy to quickly implement ad-hoc commands in an embedded system, by adding an entry in the command table and source code statements in a C/C++ switch/case statement, without worrying too much about command argument validation and predictability of command feedback to the user.

A command line comprises a *selector* token, an optional command *verb* token, and from zero or more command *parameter values*. A parameter value can be an integer numeric value or a string literal value. An empty command line is thus invalid (if such lines are to be ignored, a provision should be made in the application software).

The command line is first scanned to detect non-string tokens and string literal tokens. A string token is enclosed in double quote (") characters. The backslash (\) character escapes either the double quote character or itself. In this first scan, the selector, verb, and the integer parameter values are non-string tokens. They are sequences of adjacent printing characters (non-string tokens are separated by space or string literal tokens). Integer parameter values are valid input to the C standard functions **`strtoul`** or **`strtol`** with an undefined base (e.g. 0x1234567, -1000000000). Valid selectors and verbs are defined by the command table, except that a verb can not look like an integer parameter value.

As the next step in maintenance command processing, a matching entry is looked for in the command table for the command line. The command table structure imposes some implicit

restrictions on the possible command lines:

- the length of a command selector is limited to 7 characters,
- the length of a command verb is limited to 11 characters,
- the number of command parameter values is limited to 10.

The command table defines each valid command line syntax, with the selector, verb, number of parameter values, the type of each parameter, and a valid range for each parameter. The type of an integer parameter value is refined at the command table entry level: an integer parameter value has a maximum range from LONG_MIN to LONG_MAX (valid input string for the C standard function strtol), while an unsigned parameter value has a maximum range from 0 to ULONG_MAX. The command table allows the developer to specify a small descriptive text for the command. This text will appear as part of the detailed listing of the command table output as a response to the default command "help" (the parameter types and range are listed in a format that casts little doubt about the ad-hoc implementation strategy).

A command line match specifically occurs when a command table entry is found with

- identical selector (case-insensitive comparison)
- identical verb (case insensitive comparison), or the matched absence of a verb,
- same number of parameter values, and
- for each parameter, a match between string and non-string parameter status.

Note: Those who are familiar with the term "function overloading" will recognize this as command overloading.

Once a command table entry match has been recognized, the parameter range values are validated. For a string literal parameter, the range indication refers to the string length.

Any error encountered so far is reported as an error code value to the application software that requested the command parsing and processing. Otherwise, the command-specific processing is started. In an ABCD Proto-Kernel™ based application, the command processing may include the queueing of a processing request event entry in another task's event queue, so that the current task does not block while processing a long command.

The memory management of command line buffers and parameters is tailored to embedded systems execution environment. The command parsing software modifies the command line buffer so that the command-specific processing receives null-terminated string pointers for string literal parameters. Moreover, the command parsing software applies the backslash escape mechanism to the string arguments. The centralized heap management offered with the ABCD Proto-Kernel™ software facilitates the command-line buffer freeing once the application-specific processing is completed, irrespective of the switching of task context.

5.5 Software Diagnostics Support

5.5.1 Software Diagnostics Using LEDs

The ABCD Proto-Kernel™ source code includes provisions for system error reporting using two software-controlled LEDs, respectively LED1 and LED2. Blinking patterns are used to represents (preferably small) numbers:

- zero is represented by the LED1 off and the LED2 blinking a single cycle at a halved pace (two periods off, two period on),
- a positive number is represented by both LEDs off for two periods, followed by the LED1 blinking the number of cycles, while the LED2 remains on,
- a negative number is represented by both LEDs off for two periods, followed by the LED2 blinking the number of cycles, while the LED1 remains on.

Moreover, a positional mark allows a (preferably small) sequence of (preferably small) numbers to be represented (a large sequence of large numbers would take a long time to be represented). Once an error occurs, the system repeatedly displays the pattern made of the positional mark followed by the sequence of numbers.

For positive numbers (e.g. +3):

```
LED1 .........***...***...***
LED2 ......******************
```

For negative numbers (e.g. -4):

```
LED1 ......************************
LED2 .........***...***...***...***
```

For zero:

```
LED1 ............
LED2 ......******
```

The positional mark:

```
LED2 ......***************...
LED1 ......***************...
```

In summary, the LED diagnostic facility allows the display of a small sequence of small numbers that represent an error code displayed repeatedly once the error is detected and the software

application stopped. The use of this facility is registered in the file abcd_sw_diagnostics.h. The function **`abcd_panic_led_ns`** (also declared in the file abcd_sw_diagnostics.h) gives the application access to the above LED illumination sequences and otherwise stops the software operation.

5.5.2 Software Diagnostics Using Service History Data Recording

5.5.2.1 Basic Facilities for Service History Recording

The **`flprintf`** function uses the same calling conventions as the C standard function **`printf`**, except that it writes to the log recording capability of the FlashCnL software library (through the function **`FlashCnL_put_log`**). It can be used by tasks (but not by ISRs) to record troubleshooting traces (pedantically called *service history records*).

Once the service history is recorded in the FlashCnL memory section, the developer or field support personnel should use a log retrieval function to retrieve the service history records for later analysis. This log retrieval function should be part of a built-in maintenance application protocol either in the same application, or in a separate application that is loaded for the purpose of service history retrieval. The FlashCnL library has facilities to support a service history retrieval protocol. Otherwise, this is considered an application issue.

5.5.2.2 Processor Exceptions or Traps

If the software processor encounters an invalid instruction sequence, like an illegal memory reference or a division by zero, it usually “traps,” e.g. enters a PowerPC exception vector related to the type of illegal intruction sequence. The ABCD Proto-Kernel™ takes control when such an exception or trap occurs. If possible, the kernel software records the software failure details with the **`flprintf`** function.

5.5.2.3 Simple Programmed Software Crash

The function **`abcd_software_crash`** (declared in the file abcd_sw_diagnostics.h) allows an application software to stop the software operation and record the contents of CPU registers upon entry in the function **`abcd_software_crash`**. If the CPU architecture uses registers for parameter passing (e.g. in many RISC architectures), this is a convenient way to handle a remotely conceivable error condition that nonetheless deserves some detailed occurrence recording.

6. Implementation on the Motorola MPC8xx Processor Family

6.1 Interrupt Dispatching with the Motorola MPC8xx

6.1.1 MPC850 Interrupt Source Hierarchy

The Motorola MPC8xx processor family has an interrupt processing hardware which can be described as a three-level hierarchy, with two supplementary levels created by multiple event interrupt sources. As a general rule, these supplementary levels are not handled by the ABCD Proto-Kernel™ because they are too device-specific and application-usage-specific.

1) The exception vectors 0x500 (External Interrupt Exception) and 0x900 (Decrementer Exception) lie at the top of the hierarchy. The 0x500 exception vector is the parent of the second level of interrupts, the SIU. At this level, interrupt sources can not be masked independently; the bit MSR(EE) in the Machine State Register controls both the 0x500 and the 0x900 exception vectors. As a consequence, the Decrementer interrupt source should be considered non-maskable for practical purposes.

2) The interrupt controller in the SIU (System Integration Unit) is the second level in the interrupt dispatching hierarchy. The following interrupt sources lie at this level:
   - the external signals IRQ1 to IRQ7,
   - the periodic interrupt timer,
   - the timebase reference interrupts,
   - the real-time clock interrupts, and
   - the PCMCIA interrupt (2 sources, IP_B[0..6] external input change interrupt and IP_B7 external input monitoring interrupt).

   Moreover, the CPIC interrupt also lie at this level, as the parent of the third level in the hierarchy.

2.sup) The following second level interrupt sources have multiple events associated with them:
   - the timebase reference interrupts: the reference A and the reference B interrupts;
   - the real-time clock interrupts: the so-called "second " interrupt and the alarm interrupt;
   - the IP_B[0..6] external input change interrupt: the external input signals IP_B0 to IP_B6.

3) The CPIC (Communications Processor Interrupt Controller) is the third level in the interrupt dispatching hierarchy. Overall, 27 interrupt sources lie at this level, including 12 external interrupts (the parallel port signals PC4 to PC15) and the following 15 internal sources from the CP internal peripherals:
   - the serial ports SCC2/3 (2 sources),
   - the serial ports SMC1/2 (2 sources),

- the SPI port,
- the I2C port,
- the USB port,
- the IDMA channels 1/2 (2 sources),
- the RISC timers,
- the CP timers 1/2/3/4 (4 sources),
- the SDMA error interrupt.

3.sup) Each of the 15 internal sources from the CP has an associated "event register" (or a few event bits in a "status register"). In many cases, this event register reports more than one type of event which might be individually masked in a mask register (or in a control register). Thus, the interrupt dispatch hierarchy may be defined at this level with an application perspective in mind. Among others possibilities:
- the serial ports SCC2/3, with various protocol-specific events which may be conveniently classified into receiver events and transmitter events,
- the serial ports SMC1/2, with various protocol-specific events which may be conveniently classified into receiver events and transmitter events,
- the RISC timers interrupts, with up to 16 individual timer expiry indications.

6.1.2 Names Assigned to Regular Interrupt Sources

The interrupt sources names use the notation set forth in section 3.2.1 on page 15. The decrementer interrupt source bears the name DECR in these notations. It is non-maskable, so neither ABCD_ISRMASK_DECR_H nor ABCD_ISRMASK_DECR_L are defined.

Here is the table of interrupt signals in the level 2 in the interrupt dispatch hierarchy.

Name	Description
IRQ1	IRQ1 external input
IRQ2	IRQ2 external input
IRQ3	IRQ3 external input
IRQ4	IRQ4 external input
IRQ5	IRQ5 external input
IRQ6	IRQ6 external input
IRQ7	IRQ7 external input

Name	Description
PIT	Programmable interrupt timer
TIMEBASE	PowerPC timebase (see note)
RTC	Real-time clock (see note)
IPB06	IP_B[0..6] external input change (see note)
IPB7	IP_B7 external input monitoring
CPIC	Communications Processor grouped interrupt signal, name reserved for ABCD Proto-Kernel™ processing

Note: The sub-events of these interrupt source may be handled by the ABCD Proto-Kernel™ ISR dispatch function software, in which case the 6.1.3 section (on page 59) applies.

Here is the table of interrupt signals in the level 3 in the interrupt dispatch hierarchy.

Name	Description
PC15	Parallel I/O-PC15
USB	USB port
SCC2	SCC2 serial port
SCC3	SCC3 serial port
PC14	Parallel I/O-PC14
TIMR1	CP Timer 1
PC13	Parallel I/O-PC13
PC12	Parallel I/O-PC12
SDMA	SDMA channel bus error
IDMA1	IDMA channel 1
IDMA2	IDMA channel 2
TIMR2	CP Timer 2
RISCTMRS	RISC timer table (see note)

Name	Description
I2C	I2C port
PC11	Parallel I/O-PC11
PC10	Parallel I/O-PC10
TIMR3	CP Timer 3
PC9	Parallel I/O-PC9
PC8	Parallel I/O-PC8
PC7	Parallel I/O-PC7
TIMR4	CP Timer 4
PC6	Parallel I/O-PC6
SPI	SPI port
SMC1	SMC1 serial port
SMC2	SMC2 serial port
PC5	Parallel I/O-PC5
PC4	Parallel I/O-PC4
CPERROR	CP Error, an innocuous error reporting event for which the ABCD Proto-Kernel™ provides an adequate default ISR.

Note: The sub-events of this interrupt source may be handled by the ABCD Proto-Kernel™ ISR dispatch function software, in which case the 6.1.3 section (on page 59) applies.

6.1.3 Names Associated with Software Extension of Interrupt Controller Logic

Some of the interrupt sources in the levels 2.sup and 3.sup in the above hierarchy may be handled by the ABCD Proto-Kernel™ interrupt dispatch function, so that they appear as a regular interrupt source to the application programmer. The level of support can be either:

a) individual ISR dispatch and *individual interrupt source masking*, or
b) individual ISR dispatch but *grouped interrupt source masking*.

This extended of software support is available for:

- the PowerPC timebase reference interrupt,
- the real-time clock interrupt,

- the IP_B[0..6] external input change interrupt, and
- the RISC timers interrupt.

For sake of simplicity in the underlying operation, at most a single item in the above list can benefit from individual interrupt source masking.

Notes: 1) If the ABCD Proto-Kernel™ was implemented for the MPC860 processor, the IP_A[0..6] status change interrupt would be added to the above list.

2) For sake of determinism in hardware reaction to external events, the PowerPC interrupt architecture defines interrupt priorities which are enforced by the interrupt controller logic. The ABCD Proto-Kernel™ extends this interrupt prioritization scheme to the interrupt sources it controls by software. The relative priorities are fixed according to the event bit positions in the respective event registers indicated in the table, the bit coming first (having the highest binary weight) is assigned the highest priority.

Grouped Event Name	Individual Event Name	Description	Mask Register(Bit)	Event Register(Bit)
TIMEBASE	PowerPC timebase			
	REFA	Timebase reference A event	TBSCR(REFAE)	TBSCR(REFA)
	REFB	Timebase reference B event	TBSCR(REFBE)	TBSCR(REFB)
RTC	Real-time clock			
	RTCSEC	Second interrupt	RTCSC(SIE)	RTCSC(SEC)
	RTCALR	Alarm interrupt	RTCSC(ALE)	RTCSC(ALR)
IPB06	IP_B[0..6] status change			
	IPB0	IP_B0 external input	PER(CB_EVS1)	PSCR(CBVS1_C)
	IPB1	IP_B1 external input	PER(CB_EVS2)	PSCR(CBVS2_C)
	IPB2	IP_B2 external input	PER(CB_EWP)	PSCR(CBWP_C)
	IPB3	IP_B3 external input	PER(CB_ECD2)	PSCR(CBCD2_C)
	IPB4	IP_B4 external input	PER(CB_ECD1)	PSCR(CBCD1_C)
	IPB5	IP_B5 external input	PER(CB_EBVD2)	PSCR(CBBVD2_C)

Grouped Event Name	Individual Event Name	Description	Mask Register(Bit)	Event Register(Bit)
	IPB6	IP_B6 external input	PER(CB_EBVD1)	PSCR(CBBVD1_C)
RISCTMRS	RISC timers			
	RTMR15		RTMR(TMR15)	RTER(TMR15)
	RTMR14		RTMR(TMR14)	RTER(TMR14)
	RTMR13		RTMR(TMR13)	RTER(TMR13)
	RTMR12		RTMR(TMR12)	RTER(TMR12)
	RTMR11		RTMR(TMR11)	RTER(TMR11)
	RTMR10		RTMR(TMR10)	RTER(TMR10)
	RTMR9		RTMR(TMR9)	RTER(TMR9)
	RTMR8		RTMR(TMR8)	RTER(TMR8)
	RTMR7		RTMR(TMR7)	RTER(TMR7)
	RTMR6		RTMR(TMR6)	RTER(TMR6)
	RTMR5		RTMR(TMR5)	RTER(TMR5)
	RTMR4		RTMR(TMR4)	RTER(TMR4)
	RTMR3		RTMR(TMR3)	RTER(TMR3)
	RTMR2		RTMR(TMR2)	RTER(TMR2)
	RTMR1		RTMR(TMR1)	RTER(TMR1)
	RTMR0		RTMR(TMR0)	RTER(TMR0)

The selection of extended software support is made with conditional compilation and assembly when the ABCD Proto-Kernel™ software is built. Each of the following preprocessing symbol

```
ABCD_OPT_ISR_SW_DISPATCH_FOR_TIMEBASE
ABCD_OPT_ISR_SW_DISPATCH_FOR_RTC
ABCD_OPT_ISR_SW_DISPATCH_FOR_IPB06
ABCD_OPT_ISR_SW_DISPATCH_FOR_RISCTMRS
```

is defined to 0, 1, or 2 respectively for:

0: no extended software support for the dispatching of individual events,

1: individual ISR dispatch but grouped interrupt source masking, or
2: individual ISR dispatch and individual interrupt source masking.

If an individual event XYZ under the grouped event GRP is covered by extended software support (setting of symbol ABCD_OPT_ISR_SW_DISPATCH_FOR_GRP is 1 or 2), the preprocessor symbol ABCD_ISRDISP_XYZ is defined, and the application software should not use the ABCD_ISRDISP_GRP symbol. If the software support also includes the individual interrupt source masking (setting of symbol ABCD_OPT_ISR_SW_DISPATCH_FOR_GRP is 2), the preprocessor symbols ABCD_ISRMASK_XYZ_H and ABCD_ISRMASK_XYZ_L are also defined. Otherwise, the grouped event name should be used for interrupt disabling (symbols ABCD_ISRMASK_GRP_H and ABCD_ISRMASK_GRP_L).

6.1.4 A Singular Synchronization Requirement

Generally, the ABCD Proto-Kernel™ controls the complete mask registers (registers SIMASK, CIMR, and also RTMR whenever ABCD_OPT_ISR_SW_DISPATCH_FOR_RISCTMRS==2) in the MPC850 interrupt controller logic. With individual interrupt source masking, the ABCD Proto-Kernel™ might update the mask registers TBSCR, RTCSC, or PER register (respectively for ABCD_OPT_ISR_SW_DISPATCH_FOR_TIMEBASE==2, ABCD_OPT_ISR_SW_DISPATCH_FOR_RTC==2, and ABCD_OPT_ISR_SW_DISPATCH_FOR_IPB06==2), with an implied synchronization requirement if the application software also needs to update the same register. This is a very special case of synchronization requirements because one of the competing software sequence is part of the ABCD Proto-Kernel™.

For the PER, the synchronization requirement is addressed by forcing independent byte accesses to the two portion of the PER register, respectively PER_I06 for the register portion that is completely controlled by the kernel when ABCD_OPT_ISR_SW_DISPATCH_FOR_IPB06==2 and PER_I7 for the register portion that the application may update.

In practice, the TBSCR and RTCSC should be initialized before the ABCD Proto-Kernel™ interrupt system becomes operational, and then not used by the application. In the very exceptional case where an application would need to update either of these registers, the application software should enclose the register update operation between calls to **`extern void Set_EID(unsigned long);`** and **`extern void Set_EIE(unsigned long);`**, thus totally preventing interrupts from interfering with this software operation.

6.2 Assembler Language ISRs Programming

6.2.1 ISR Entry and Epilogue

6.2.1.1 ISR Entry Conditions

Upon entry in an ISR written in assembler, the following registers are available for use by the ISR: **XER**, **CR**, **LR**, **%R24** to **%R26**. The register **%R25** points to the ISR table entry for the interrupt source. The ISR may use this **%R25** register pointer (perhaps moved to **%R1** as explained below) with the displacements macros of the form ABCD_ISRTABLE_..., but the ISR needs not preserve this register.

The ISR must preserve the additional registers it might use. When performance considerations are critical, there are two execution patterns for which a recommendation can be made. If the ISR epilogue returns to the interrupted context with a high probability and its latency is to remain small, the ISR should preserve the minimum set of registers. If the ISR epilogue enters the ABCD scheduler with a high probability, it is better to preserve the whole set of registers.

The ISR may preserve additional general purpose registers, from 1 to 29 of them, plus the register **CTR**. This is done through the preprocessor macro ABCD_SAVE_REGS(NB_REGS,R25_TO_R1_COPY_FLAG). The first argument NB_REGS to this macro is the number of general purpose registers to be saved. The second argument R25_TO_R1_COPY_FLAG must be set to 0 when saving less than 6 registers. If set to non-zero, this argument R25_TO_R1_COPY_FLAG means that the pointer value originally in **%R25** is moved to **%R1** during the register saving sequence, otherwise the contents of **%R25** is lost by saving 6 or more registers.

The order in which the registers are preserved is specified in the table below. The order in which the general purpose registers are made available by the preservation sequence is not natural. The table contents shows that the register **CTR** is available if and only if at least 5 general purpose registers are preserved and that requesting the preservation of 6 registers has the same effect as requesting 7 of them.

Number of General Purpose Registers to be Preserved	General Purpose Register Incrementally Preserved	Additional Register Preserved
0		**XER**, **CR**, **LR**, **%R24** to **%R26**
1	**%R31**	
2	**%R30**	
3	**%R29**	
4	**%R28**	

5	**%R27**	**CTR**
6	**%R1**	**%R0**
7	**%R0**	
8	**%R23**	
9	**%R22**	
10	**%R21**	
11	**%R20**	
12	**%R19**	
13	**%R18**	
14	**%R17**	
15	**%R16**	
16	**%R15**	
17	**%R14**	
18	**%R13**	
19	**%R12**	
20	**%R11**	
21	**%R10**	
22	**%R9**	
23	**%R8**	
24	**%R7**	
25	**%R6**	
26	**%R5**	
27	**%R4**	
28	**%R3**	
29	**%R2**	

For instance, a classical ABCD Proto-Kernel™ ISR simply inserts an event code in an event queue, and thus requires the additional registers **%R31** and **%R30**:

6.2.1.2 ISR Epilogue

ABCD_ISR_EPILOGUE(TRIGTSK_LEVEL,NB_REGS,LOCAL_LABEL_CHARS)

The macro ABCD_ISR_EPILOGUE is a C preprocessor macros that also relies on assembler directives (e.g. conditional directives **.if** and **.endif**) to generate the appropriate code sequences from the macro arguments such as TRIGTSK_LEVEL.

When expanded, the macro ABCD_ISR_EPILOGUE consists of three assembler instruction sequences:

1) at the exit point in the ISR application processing, the *epilogue decision sequence*,

2) followed immediately by the *rfi sequence* (rfi refers to the return from interrupt assembler instruction mnemonic), and

3) the *scheduler entry sequence* which is introduced by a label used in the epilogue decision sequence. The scheduler entry sequence requires a unique assembler label name to be defined as the label introducing the scheduler entry sequence.

As explained for the ABCD Proto-Kernel™ in general (see section 3.2.5 on page 20), the argument TRIGTSK_LEVEL selects the ISR epilogue variant to be expanded:

- ABCD_KERNEL_SCHED_CONTEXT_COUNT+ABCD_TASK_COUNT or higher values (e.g. SHRT_MAX), implies no scheduler entry sequence,

- from ABCD_KERNEL_SCHED_CONTEXT_COUNT to the ABCD_KERNEL_SCHED_CONTEXT_COUNT+ABCD_TASK_COUNT-1 constant inclusive, implies a comprehensive macro expansion, and

- zero (symbolically ABCD_KERNEL_SCHED_CONTEXT_COUNT-1), implies no rfi sequence.

The argument NB_REGS to the ABCD_ISR_EPILOGUE must match the argument NB_REGS to the macro ABCD_SAVE_REGS at the ISR entry, or zero if there was no such macro call.

The argument LOCAL_LABEL_CHARS to the ABCD_ISR_EPILOGUE should contain one or more characters (among letters, digits, and underscore, not quoted) that can differentiate the various

expansions of the ABCD_ISR_EPILOGUE in the same source file. These characters are concatenated in the macro expansion to make unique local assembler labels.

6.2.2 Services to ISRs Written in Assembler

6.2.2.1 Event Queue Insertion by ISRs Written in Assembler

The C preprocessor macro QUEUE_GET_IN_PTR **(LOCAL)** generates the assembler instructions for queue entry insertion. The source file in which this macro is used must also make one call to the macro DECLARE_QUEUE_EXCEPTION_PROCESSING to generate the assembler instructions for a required local routine (this single macro call must be made in an instruction assembler section, outside of any other routine or ISR).

The macro QUEUE_GET_IN_PTR is declared in the file abcd_evqueues.h. It expects a pointer to the ABCD Proto-Kernel™ queue object in the register **%r24**, and returns an entry pointer in register **%r25**. The registers **%r26**, **%r30**, **%r31**, **cr0**, and **lr** can be changed by the macro QUEUE_GET_IN_PTR.

6.2.2.2 Interrupt Masking Service

The details of a service to disable or enable interrupts, compatible with the critical real-time arena internals, is currently neither specified nor implemented for ISRs written in assembler. This is an ABCD Proto-Kernel™ implementation deficiency.

6.2.2.3 Timestamping Service in ISRs Written in Assembler

The timestamping service in ISRs written in assembler is supported by preprocessor macros that properly sample the timer registers upon which the **clock_t** type is based. These timers are organized in pairs, either the TBU / TBL pair when the PowerPC timebase facility is used for the **clock_t** type, or the RTC / RTCSEC pair when the MPC8xx real-time clock facility is used for the **clock_t** type. The relevant preprocessor macros are ABCD_ISR_TIMEBASE_SAMPLE and ABCD_ISR_RTC_SAMPLE, both declared in the file abcd_interrupts.h. These macros take three arguments: a high register designation, a low register designation and a scratch register designation.

Once this raw data is brought to a task context, usually through the queueing mechanism, a scaling operation has to be performed. The scaling operation is performed with the functions **Get_scaled_timebase** and **Get_scaled_rtc**.

6.3 CPU Context Save Areas

6.3.1 Task Interrupted by an Asynchronous Event

Here is the sequence of task context registers when they are stored in a memory structure for tasks that are pre-empted from the CPU by an asynchronous event:

- %R2 to %R23 inclusive,
- %R24 containing a copy of %R1,
- %R25 containing a copy of %R0,
- %R26 containing a copy of CTR,
- %R27 containing a copy of XER,
- %R28 containing a copy of CR,
- %R29 containing a copy of LR,
- %R30 containing a copy of SRR0,
- %R31 containing a copy of SRR1, and
- %R24 to %R31 inclusive.

6.3.2 Task Interrupted Synchronously

When a task relinquishes the CPU synchronously, the EABI conventions are used to implement a more efficient task context switching. The following registers are saved:

- %R2
- %R1
- %R11 containing a copy of LR
- %R12 containing a copy of CR
- %R13
- %R14 to %R31

6.4 Allocation of Resources

6.4.1 CPU Registers

SPRG0

Used in the PowerPC exception processing sequence, while MSR(RI) is zero.

SPRG1

Reserved.

SPRG2
SPRG3

Reserved for reset and other destructive exception processing.

%R13

Holds the IMMR pointer. The lower 16 bits are zero, each of the two upper bytes are non-zero, and bit 15 (weight 2^{16}) is one.

%R2

Holds a pointer to the task local data, valid when MSR(EE) is one and during execution of an ISR written in C. Upon entry of an ISR written in assembler, holds the pointer to the task local data of the interrupted context.

6.4.2 Machine State Register Bits

Bit	Name	Name	Usage in ABCD Proto-Kernel™
13	POW	Power management enable	Application control (planned enhancement)
15	ILE	Exception little-endian mode	fixed, zero
16	EE	External interrupt enable	kernel control
17	PR	Privilege level	fixed, zero
18	FP	Floating-point available	fixed, zero
19	ME	Machine check enable	kernel control
21	SE	Single-step trace enable	fixed, zero
22	BE	Branch trace enable	fixed, zero
25	IP	Exception prefix	kernel control, i.e. may be changed during startup sequence
26	IR	Instruction address translation	fixed, zero
27	DR	Data address translation	kernel control (i.e. linked to data cache enabling)
30	RI	Recoverable exception	kernel control
31	LE	Little-endian mode enable	fixed, zero

6.4.3 PowerPC Exception Vectors

Address	Section Name	Exception Type	Exception Handling Category
0x100		System Reset	Non-maskable interrupt
0x200	Machine_Check	Machine Check	System crash with diagnostic
0x500	External	External	Interrupt processing
0x600	Alignment	Alignment	System crash with diagnostic
0x700	Program	Program	System crash with diagnostic
0x900	Decrementer	Decrementer	Interrupt processing
0xC00	System_Call	System Call	System crash with diagnostic
0xD00	Trace	Trace	System crash with diagnostic
0x1000	Software_Emulation	Implementation-Dependent Software Emulation	System crash with diagnostic
0x1100	I_TLB_Miss	Implementation-Dependent Instruction TLB Miss	System crash with diagnostic
0x1200	D_TLB_Miss	Implementation-Dependent Data TLB Miss	System crash with diagnostic
0x1300	I_TLB_Error	Implementation-Dependent Instruction TLB Error	System crash with diagnostic
0x1400	D_TLB_Error	Implementation-Dependent Data TLB Error	System crash with diagnostic
0x1C00	Data_Breakpoint	Implementation-Dependent Data Breakpoint	System crash with diagnostic
0x1D00	Instr_Breakpoint	Implementation-Dependent Instruction Breakpoint	System crash with diagnostic
0x1E00	Periph_Breakpoint	Implementation-Dependent Peripheral Breakpoint	System crash with diagnostic
0x1F00	Nonmask_Dev_Port	Implementation-Dependent Nonmaskable Development Port	System crash with diagnostic

6.5 Communications Processor

In the Motorola MPC8xx processor family, the communications processor transmit and receive processing use a list of buffer descriptors (BDs) in a hardware-specific queuing mechanism. The hardware distinctive characteristic is the circular BD management in the coupling arrangement

between the core CPU and the co-processor called communications processor (CP). The reader is referred to the document [MPC850_UG] for detailed reference information. This scheme was introduced by Motorola with the MC68302 processor a while back, and has been retained and upgraded in the MC68360 processor and the MPC8xx and the MPC82xx processor families.

Strictly speaking, the ABCD Proto-Kernel™ software need not be concerned with the MPC850 CP. However, there are a few perspectives on the CP that are relevant to the embedded software developer:

- The trace mechanism is a useful addition to the ABCD Proto-Kernel™ which relies on a MPC8xx serial communications channel for transmitting trace packets (see section 5.4.1 on page 51).

- An application software may require the use of other MPC8xx serial communications channels and/or the sharing of the communications channel used by the trace mechanism.
  Note: The PPCMB/850 embedded loader is indeed an application that shares the communications channel with the trace mechanism.

- The handling of the MPC8xx CP in the software development process is a multi-faceted issue, including
  - the software creation procedures (see the document [ABCD_LINK_N_LOAD]),
  - the system initialization sequence (see the document [PPCMB850_INIT]).

6.5.1 Communications Processor Configuration

For the MPC8xx communications processor, some configuration aspects are decided at compile-time, mostly using preprocessor symbols in the the files `abcd_cfg_defs.h`, `abcd_config.h`, and `abcd_cfg_rules.h`, but also through the file `mpc8xx_clk_cfg.txt` that is input to a development utility for processor clock distribution configuration (both mechanisms are further documented in the document [ABCD_LINK_N_LOAD]). Here is a list of configuration elements that can be decided at software creation time:

- Whether a communications channel is used or not.

- For the SCC2/3 and SMC1/2 communications channels, the communications processor protocol used.

- The static allocation of BDs.

- The assignment of BRGs (Baud Rate Generators) to SCC2/3 and SMC1/2.
  Note: With the MPC850 with most typical system designs, the four BRGs should be in

sufficient number for the serial channels that may need a BRG (USB, SMC1/2, SCC2/3 RxClk, SCC2/3 TxClk). This is less certain for other MPC8xx processors, and this suggests a dynamic allocation of BRGs for asynchronous baud rate generation (e.g. to match baud rate configuration information stored in the flash).

- Whether the RISC timers are used.
- Whether the RISC microcode patch is implemented (required whenever SCC2 or SCC3 uses the Ethernet protocol).
- The reservation of dual-port memory for downloadable microcode patch.
- The pre-computed clocking parameters.
- Many channel-specific and/or protocol-specific configuration items, e.g.
  - the maximum idle parameter for SMC1/2 and SCC2/3 asynchronous configuration,
  - the control characters table for SCC2/3 asynchronous protocol configuration,
  - the CRC parameters for SCC2/3 protocols using a CRC.

Many other aspects of the CP operations are remaining as run-time issues. This includes for instance the MRLB (Maximum Receive Buffer Length) parameter for each serial channel, that is often related to system-wide buffer management.

6.5.2 Handling of Buffer Descriptors

The information associated with outstanding BDs is kept in an ABCD Proto-Kernel™ queue (the "*BD queue*"). The queue entry count must match the number of BDs (that is, with N BDs, the queue constructor has N+2 entries). This control scheme is identical for the transmit and receive directions.

The synchronization mechanisms for serial communications are two-tiered:

Tier	Initiation of Serial Communications	Detection of Serial Communications Completion	Completion of Serial Communications

at the CP interfacing level	when the BDs are inserted by setting the Empty/Ready bit in the BD status, the CP is signaled to process the BD	Empty/Ready bit cleared	the core is signaled to process the BD, often through an interrupt signal
at the ABCD Proto-Kernel™ scheduling level	queue entry insertion	queue not empty condition	queue entry extraction

The above table details the following overall principles:

> A BD queue not empty condition indicates that some outstanding BDs are expected to be returned. The oldest entry in the queue is the next one to be returned, and the associated BD must be queried for detection of serial communications completion.
>
> Once the communications processor returns a BD, the oldest queue entry should be extracted. This is done in a task called the completion task. The completion task's event queue is distinct from the BD queue.
>
> To post a new buffer, the software first locates the next queue entry for insertion. If a queue full condition is detected at this point, the maximum number of outstanding BDs has been reached. If and when the post to the CP is completed, the queue entry insertion can be finalized.

Three models can be envisioned for kernel serialization support of serial communications. Each of them is based on ISR signaling of BD processing completion by the CP (the ISR inserts an entry in the completion task's queue). Also, the indication that a BD processing is complete is made of the queue-not-empty condition *and* the clearing of the relevant Empty/Ready bit by the CP.

The insertion of a BD in the communication process is done as the *initiation sequence* comprising the following steps:

a) **`mpc8xx_bd_queue_SMC1_tx.get_in_entry();`**
b) record the current value of the **`next_bd`** index for steps d) and e)
c) advance the **`next_bd`** index in the BD circular array;
d) fill the BD pointer field (and length field in the case of transmission);
e) set the BD status word, including the Empty/Ready bit;
f) fill other required fields in the **`mpc8xx_bd_queue_SMC1_tx`** entry, if any;
g) **`mpc8xx_bd_queue_SMC1_tx.advance_in();`**

The corresponding processing for extraction of a completed BD is the *completion sequence* comprising the following steps:

h) test **`mpc8xx_bd_queue_SMC1_tx.is_empty()`**, and performs the steps i) through l) only if it not empty;
i) record **`mpc8xx_bd_queue_SMC1_tx.get_out_entry();`**
j) test the corresponding Empty/Ready bit (a cleared Empty/Ready bit is an indication of a BD returned by the CP), and perform the steps k) and l) only if the BD is returned;
k) process the returned BD;
l) **`mpc8xx_bd_queue_SMC1_tx.advance_out();`**

The completion sequence is done by the completion task when it receives a channel event notification entry in its task queue.

The synchronization issues with the initiation and completion sequences must consider the following two issues:

1) Say tasks A and B compete for the CPU during the initiation sequence. The task A perform step a) and is then pre-empted by the task B which then performs the steps a) through c) before the task A gets an opportunity to complete the step c). This is a classical problem solved by mutex protection, surrounding initiation steps a) through c). However, if the task B is replaced by an ISR, a temporary interrupt source masking strategy is needed.
2) There is no way to prevent the CP from completing the BD processing between steps e) and f), even if this sequence of events is very unlikely in most practical cases. Thus, some synchronization provision must protect the start of the completion sequence, so that the immediate reaction to the initiation step e) is delayed for the initiation step f) to complete. This can be achieved with a mutex protecting both initiation steps e) through g) and the completion step h). Again, interrupt source masking may also be considered.

If a single task does all channel processing (in either or both directions or transmission), the initiation and completion sequences are implicitly serialized. Otherwise, the straightforward solution is to use a single mutex for the two synchronization issues, protecting both initiation steps a) through g) and the completion step h). Other schemes may be designed. When interrupt source masking is considered, attention must be paid to the interactions that are not obvious when investigating the source code, and to the possibility for the completion task to be awakened by events not related to the communications channel.

6.5.3 Naming Conventions

It may be convenient to use normalized field names in a BD queue entry class or structure definition. The field name **`bd_pt`** should be used for a pointer to the CP BD. The field names **`raw_buf_pt`** and **`recyc_fn`** should be used for pointers to respectively a “raw buffer” and a function that recycles this raw buffer. The buffer actually processed by the CP

(**bd_pt->buffer**) might be the same as the raw buffer (**raw_buf_pt**), but need not be (a small offset is conceivable if the raw buffer contains a header that should be ignored by the CP). Here is the start of an acceptable type declaration for a BD queue entry:

```
class a_bd_queue_entry_type_declaration
{
public:

   /* Field name for normalized processing of BDs returned by the
     CP. */
   struct mpc8xx_buf_desc_str *bd_pt;

   /* Field names for normalized processing of buffer memory
     recycling once the buffer returned by the CP is no longer
     used. */
   void *raw_buf_pt;
   void(*recyc_fn)(void*raw_buf_pt);

   /* ... other fields ... */
```

Otherwise, the details of a BD queue entry type is application-specific.

It may also be convenient to consider the BD queue as an application entity with a name of the form **mpc8xx_bd_queue_CCC9_dd** where **CCC** is the channel type designation (**SCC**, **SMC**, **SPI**, **I2C** or **USB** for USB endpoint), **9** is the channel number, if any, within the **CCC** type designation (as in **SCC3** or **USB4**), and the **_dd** suffix is either the **_rx** communications direction, or the **_tx** communication direction.

6.6 Low Power Management

The low power management support is an ABCD Proto-Kernel™ useful addition (it is not an intrinsic part of The ABCD Proto-Kernel™ design). In microprocessor systems, low power management is usually based on clock frequency division (or clock halting) in the some of the microprocessor functional blocks. Thus, low power management may have adverse effect on the normal operation of any time-related peripheral (e.g. timers, but also serial communications channel that depend on baud rate generators or internal clock for DPLL-based clock recovery).

In the ABCD Proto-Kernel™ implementation, the low power modes are entered in an infinite loop implemented in the lowest priority task, in the source file abcd_utils_pwrman_idle.cpp. The application task **abcd_utils_pwrman_idle_task** must be part of the application software task list declarations (see section 3.3.3 on page 24). Any *serviced ISR* triggers an exit from the

low power mode. After the application processing of the ISR event queue entries, the lowest priority task re-enters the low power mode. In this explanation, a serviced ISR is an interrupt source that is recognized by the processor in the current low power mode. An exception is the decrementer exception that is handled differently by the processor in some of the low power modes. For details, see the document [MPC850_UG].

6.6.1 Idle CPU Power Saving

The aim of the CPU power saving is to reduce power consumption with insignificant or minimal impact on system performance. This mandates adequate responsiveness on communications channel activity and other interrupt sources. With the Motorola MPC8xx processor family, this is achieved with the halting the clocks to the CPU (that is, entering one of the two low power *doze* modes) and letting the CP (Communications Processor) operate at a reduced clock frequency that is still adequate for the highest speed communications channel (that is, entering the low power *doze low* mode with an optimized value for **SCCR(DFNH)**). The idle CPU power saving is almost fully transparent to the application software and requires no hardware features outside of the PPCMB/850 hardware.

The use and implementation of the idle CPU power saving is controlled with the preprocessor symbol ABCD_LOW_PWR_MAN_IDLE_SELECT that can take three possible values:

> ABCD_LOW_PWR_MAN_IDLE_NORMAL_HIGH, when the idle CPU power saving is not used,
>
> ABCD_LOW_PWR_MAN_IDLE_DOZE_HIGH, when the idle CPU power saving is to use the MPC8xx *doze high* low power mode,
>
> ABCD_LOW_PWR_MAN_IDLE_DOZE_LOW_CPM_HIGH, when the idle CPU power saving is to use the MPC8xx *doze low* low power mode, with the communications processor using the higher clock frequency when active (i.e. **SCCR(CRQEN)** is set),
>
> ABCD_LOW_PWR_MAN_IDLE_DOZE_LOW, when the idle CPU power saving is to use the MPC8xx *doze low* low power mode, with the communications processor using the lower clock frequency when active (i.e. **SCCR(CRQEN)** is cleared).

6.6.2 System Sleep Power Saving

The sleep mode is triggered by the application software when it detects a *sleep condition* where the system can or should consume the very minimal power, without normal reaction to every external event sources, with a periodic system wake up to sense the sleep condition, so that a form of auto-restart capability is provided (with an implied reduced responsiveness to external

events). By design, the sleep mode reduces system responsiveness to achieve minimal power management, so a system inactivity timer may be the source of sleep condition. Another possible source for the sleep condition is an external indication that a main power source has been lost and a backup source is being used, for the purpose of preserving the dynamic system state in volatile memory.

With the Motorola MPC8xx family, there are three possible low power modes: *sleep*, *deep sleep*, or *power-down*. The PPCMB/850 hardware supports only the sleep mode.

The use and implementation of the sleep mode is controlled with the preprocessor symbol ABCD_LOW_PWR_MAN_SLEEP_SELECT that can take three possible values:

> ABCD_LOW_PWR_MAN_SLEEP_NONE, when the sleep mode is not used,
>
> ABCD_LOW_PWR_MAN_SLEEP_YES, when the sleep mode feature is available for application use.

When available for use, the sleep mode is requested by setting the global flag variable **`abcd_utils_pwrman_sleep_request`** to a non-zero value (declared in the file abcd_utils_wait_fncs.h). Some of the the timer interrupts and the IRQ1 to IRQ7 signal interrupts may be serviced in the sleep mode (based on some MPC8xx configuration settings). Long-term exit from the sleep mode is controlled by the application servicing of these interrupt sources, by storing 0 to the global flag variable **`abcd_utils_pwrman_sleep_request`**.

Annex A. Partial List of Acronyms

ABCD
: a) interrupt dispatching, b) fixed priority scheduling, c) mutual-exclusion semaphores, and d) a queuing mechanism

API
: Application programming interface

CP
: Communications processor

CPLD
: Complex programmable logic device

CPU
: Central processing unit

CRC
: Cyclic redundancy check

DMA
: Direct memory access

FPGA
: Field-programmable gate array

GNU
: GNU is not Unix

GPL
: GNU public license

ISR
: Interrupt service routine

MMU
: Memory management unit

RAM
: Random access memory

RISC
: Reduced instruction set computer

ROM
: Read-only memory

RTC
: Real-time clock

RTCA

SIU
: System Integration Unit (MPC8xx acronym)

SPI
: Small peripheral interface

USB

Universal serial bus

Annex B. Typical Embedded Software Application Design

B.1 Overview of the Guardian Automated Vehicle System

An hypothetical embedded system project, the "*Guardian*", is used as a basis for the software examples in this document. The system is a miniature automated vehicle with three wheels, moving like a wheelchair.

Note: This section is far from finished. It describes a fictitious application that would be typical of an ABCD Proto-Kernel™ application. In order to complete this document section, source code skeletons and fragments should be interspersed in the text, and redundant material should be removed.

To make the project challenges as close to a real-life project, an actual mission is assigned to the Guardian, basically intended for surveillance and fire casualties prevention in handicapped residences:

- in normal operation, at night when little personnel circulation is expected in corridors, the Guardian goes to the rounds around the corridors and senses infrared radiations and other environmental parameters to generate events and alarms accordingly,
- in emergency mode, the Guardian also goes to the rounds, actively seeking obstacles on the floor, reporting observations in a more or less continuous mode.

To assist its motion, the Guardian microprocessor flash memory contains a digitized floor plan of the corridors. The Guardian has a short range radio for synchronous data transmission, using a frequency reserved for emergency signals with a simple ad-hoc protocol. The details of its sensors are given below. A Guardian docking station provides the battery charger function.

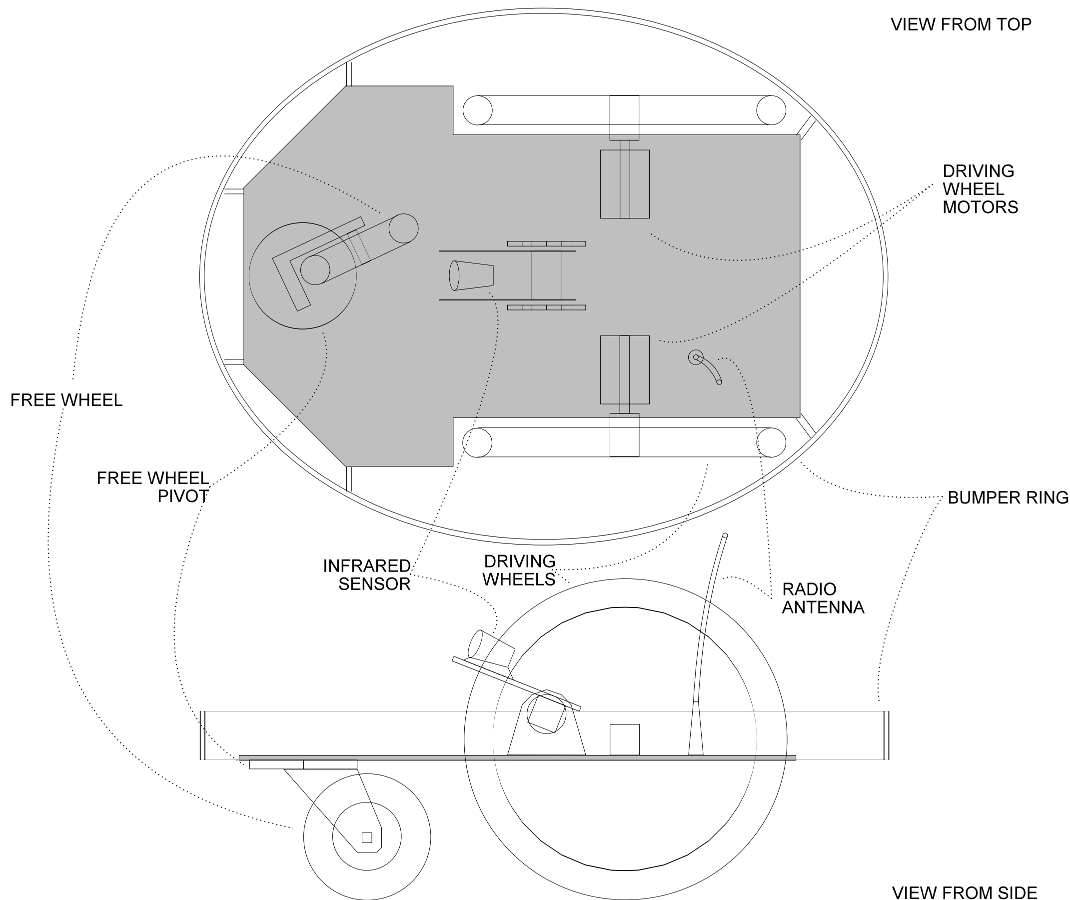

Figure 3. The Guardian Automated Vehicle Project

Here are some design considerations:

- The production cost target for the complete system is to be reasonable.
- The device should be small enough so that a firefighter walking the corridor with boots can easily kick it when it is on the way.

B.2 Preliminary Hardware Specifications

Motion subsystem:

- Each of the two driving wheels has its own motor drive electronics receiving a pulse-width modulation signal, a direction signal, and a brake signal.
- The free wheel has no motor, and no electronic control.

Navigation and attitude sensors:

- On each of the two driving wheels: quadrature encoders.
- On the free wheel pivot: a quadrature encoder with an index signal active when the free wheel is parallel with the driving wheels.
- Two accelerometers sensing instantaneous horizontal acceleration at 90 degrees angle.
- One optional accelerometer sensing vertical instantaneous acceleration, required if the Guardian has to climb and descend wheelchair ramps.
- For prototyping units only, a fiber optics gyroscope subassembly.

Elevation pointer subassembly for directional sensors:
- The elevation pointer motor has its own motor drive electronics receiving an on/off digital signal, and a direction signal.
- A quadrature encoder gives feedback on the elevation pointer subassembly angular position.
- Two limit switches inputs to the microprocessor indicate that the elevation subassembly is at either of its extreme angular positions.

Sensors (electronic interfacing not specified at this point):
- A directional infrared sensor (installed on the elevation pointer subassembly).
- A smoke detector.
- A local air visibility/opacity/turbulence detector.
- Temperature sensor.
- A number of pressure switches or sensors on the outside of the Guardian oval bumper, e.g. from 8 to 32 of them.

Communications:
- The radio uses a synchronous serial transmission protocol so that data is transferred in binary data packets which are protected by a CRC for transmission error detection.
- A development serial port.
- A serial port connected to the fiber optics gyroscope subassembly on prototyping units.
- An infrared communications port for communications with the Guardian docking station.

B.3 Embedded System Software Organization

B.3.1 Typical ABCD Proto-Kernel™ Task Organization

The embedded software is organized as a collection of dedicated tasks (each task has a dedicated function in the embedded application software). Each task is a polling loop that processes events that are closely related to the task function. The task receives event notifications through an 'event queue.' When no event is available for a task, the ABCD Proto-Kernel™ scheduler gives the CPU to a lower priority task that is ready to run.

The basic task structure is shown in the program skeleton below. This shows the details of event queue declaration. The array **sample_applic_queue_data** is never referenced directly outside of the event queue declaration. Variations from this basic program structure are possible, but the basic idea is highly suitable to the portions of an embedded system for which the interrupt responsiveness is important.

```
/* sample_applic.cpp */

#include "abcd_incl.h"
#include "mpc8xx_core.h"
#include "spr.h"
#include "abcd_kernel.h"
#include "abcd_evqueues.h"

/* Declarations of interest for other source files */

struct event_str
{
     int  event_code;
     /* ... event details data fields ... */
};

extern class abcd_ev_queue<struct event_str>
                    sample_applic_queue;

/* Declarations for this source file */

static struct event_str sample_applic_queue_data[25];

class abcd_ev_queue<struct event_str>
     sample_applic_queue
          (sample_applic_queue_data
          , sizeof(sample_applic_queue_data)
           /sizeof(sample_applic_queue_data[0])
          );
```

```
void sample_applic_task(void)
{
     /* ... task-specific initializations ... */
     for (;;) {
          struct event_str *event_pt;

          abcd_wait_one_queue(&sample_applic_queue);

          event_pt=sample_applic_queue.get_out_ptr();

          switch (event_pt->event_code) {
            case 1:
               /* ... processing for event code no. 1 ... */
               break;
            /* ... processing for other event codes .... */
          }

          sample_applic_queue.advance_out();
     }
}
```

In the hardware design, the allocation of external signals to microprocessor pins, and the functions allocated to external logic implemented in CPLDs (Complex Programmable Logic Device) and FPGAs (Field Programmable Gate Array), determines the interrupt capability of each individual signal. For instance, it is reasonable to route a limit switch signals to a microprocessor pin with interrupt capability so that the task that controls the corresponding motor can be alerted by an ISR. The details of the Guardian hardware design is not specified in this document. In the following description of Guardian software tasks, it is implicitly assumed that the relevant external events can trigger some ISRs.

B.4 List of tasks

B.4.1 High Priority Task Group

The high priority task group includes the hard real-time portions of the software. These are functions for which very short deadlines must be met for the proper operation of the system.

B.4.1.1 System Error Monitor Task

This task receives notifications of catastrophic error conditions, either directly from some ISRs or from other tasks. In this context, a catastrophic error is something that threatens the Guardian hardware and/or software integrity, not an alarming environmental condition detected by the Guardian sensors. In normal operation, this task does not use any noticeable portion of the CPU time in spite of its high priority: it basically waits for event notifications.

B.4.1.2 Motor Control Loop for the Guardian Motion

The motor control loop task implements a digital PID loop that periodically controls the PWM output to the two motors on the Guardian driving wheels. This task receives a timer notification at a frequency that is an exact divisor of 4800 Hz, but no greater than 300 Hz (these numbers are for illustrative purposes only, take 250Hz as an example). The digital PID loop main input is the periodic sampling of a wheel position counter, from which an instantaneous speed measurement can be deduced (actually an average speed over a 4ms period).

The wheel position counter is implemented in hardware as a counter register in a CPLD that decodes the quadrature encoder signals from each wheel. This is a wrapping counter. The sampling of the two counters is made in the 250Hz timer ISR so that the time jitter on the sampling is minimal. The motor control lop task receives a 250Hz timer tick event from this ISR with the sampled positions (and perhaps the precise sampling time) as event detail data.

The current command to the motor control loop task is an indication of desired speed and acceleration for each of the two driving wheels. This information is contained in a global data structure for which a mutex reservation is required before accessing (either for read or write). The mutex synchronization mechanism is preferred in this case to the queueing mechanism because a speed and acceleration command should overwrite any existing command, irrespective of whether the existing command was ever effected or not.

B.4.1.3 Infrared Sensor Data Acquisition Task

This task is responsible for time-critical data acquisition in the Guardian automated vehicle. Its design is dictated by the interfacing requirements of the actual sensors installed on the vehicle (even this task's relative priority might be adapted to the system electronics). Because the infrared input data is the foremost source of information for the Guardian mission and the infrared electronics should be state-of-the-art, it can be anticipated that the data rate and/or processing latency be challenging.

B.4.2 Intermediate Priority Task Group

In the intermediate priority task group, we will find typical embedded software functions that are generally I/O bound and that are conveniently structured as a polling loop. The typical deadlines of these tasks are longer and/or not as critical as for the high priority task group. Nonetheless, in some of these tasks must be protected against excessive CPU delays caused by relatively higher priority tasks, so they are real-time tasks in this respect.

B.4.2.1 Instantaneous speed, turn, and elevation monitoring task.

This task maintains the overall vehicle position and motion information, and controls the elevation pointer subassembly angular position and motion. This is where the short term vehicle path and elevation pointer positioning are controlled. Any input data that can assist the determination of vehicle speed, direction, and inclination should be made available to this task, preferably affixed with a precise timestamp of the input data validity. This includes proximity or pressure sensors along the vehicle bumper, as indications of obstacles along the vehicle route. Various signaling methods are available to bring the input data to this task:

- A global structure (protected with a mutex) that contains input data and related information, e.g. the most recent and the previous sensor reading with their timestamp.
- A sensor data event inserted in this task's queue by another task that is in a better position to validate and circumstantiate the raw sensor input data.
- ISR-provided events in this task's queue, in which case the sensor data validation is expected to be minimal.
- Direct reading of the sensor input data by this task.

Logically, this task provides the following outputs:

- The motor control loop command.
- Obstacles encounters data for the route navigation task and the data synthesis task.

This task is also assigned the elevation pointer subassembly position control function. This should be a fairly simple control function (turn on the motor) assisted by a dedicated timer ISR (turn off the motor). This is an example of a software function that can conveniently be assigned to almost any task in the intermediate priority task group.

B.4.2.2 Radio Receiver Communications Task

This task processes the data packets received through the radio serial data link. The ISR for the radio receiver channel is the primary source of event entries in this task's event queue. This task structure is a good example of a serial communication receiver task when the input data packets can encapsulate various application-specific messages with a

packet header containing fields such as a destination address, a service access point identifier (e.g a TCP/IP port number), a protocol identifier, and the like.

This task validates the received data packets and ignores those that are not addressed to this specific Guardian device or whose protocol is not supported. It is then able to action the data packets, which would often means inserting an entry in some other task's event queue.

The memory buffers occupied by a received data packet is best recycled after the packet processing is completed. When the data packet triggers an entry insertion in some other task's event queue, the entry detail data should contain three pointers:

- a pointer to the application-specific data within the received data packet,
- a pointer to the memory buffer holding the received data packet,
- a pointer to a function to be called to recycle the received data packet.

The packet processing by the other task uses the first pointer. The other two pointers are used for the memory buffer recycling. This arrangement allows the same application data packets to be received from other serial channels, transparently for the application task.

Generally, the software management of receive buffers in an embedded system must be designed to meet communications hardware specifications and eliminate the need to copy the receive buffer contents. Determinism in the allocation of buffers is also important, as is the separation of memory resources allocated to each channel (that is, a memory exhaustion condition on one channel should not stop other channels). The radio receiver communications task gives an opportunity to demonstrate the suitability of the ABCD Proto-Kernel™ facilities for this management of receive buffers. The native C language **`malloc`**/**`free`** functions or the C++ language **`new`**/**`delete`** operators are often inadequate.

B.4.2.3 Alarms Computation Task

The alarms computation task determines when it is appropriate to send an alarm indication through the radio transmitter. It is a data synthesis task intended to reduce the likelihood of false alarms, e.g. from an excessive heat sensor input when the Guardian is near a heating appliance included in the digitized floor plan. The alarms task priority need not be any much higher than the radio transmitter task priority since the alarm indications are pointless if they can not be sent. Otherwise, the organization of the alarms computation task is left as an open issue.

B.4.2.4 Radio Transmitter Communications Task

Basically, the radio transmitter task handles the transmitter related ISR generated events

and controls the "transmit opportunity" on the radio serial channel. This task has two queues, a "event queue" for ISR events that must be processed without waiting, and a "task request queue" that is considered only when a transmit opportunity is present. Thus, the transmitter task structure is organized around the **`abcd_wait_two_queues`** kernel function. While the transmitter is busy fulfilling a request, the task calls the **`abcd_wait_one_queue`** function to handle the transmitter-related events from its "event queue". When a transmit opportunity occurs, either a) an entry is gotten out of the "task request queue" or b) a call to the **`abcd_wait_two_queues`** function puts the task in a waiting mode for either a request or an ISR generated events.

```
/* sample_2qs_applic.cpp */

#include "abcd_incl.h"
#include "mpc8xx_core.h"
#include "spr.h"
#include "abcd_kernel.h"
#include "abcd_evqueues.h"

/* Declarations of interest for other source files */

struct event_str
{
     int  event_code;
     /* ... event details data fields ... */
};
extern class abcd_ev_queue<struct event_str>
                    sample_event_queue;

struct request_str
{
     int  request_code;
     /* ... request details data fields ... */
};
extern class abcd_ev_queue<struct request_str>
                    sample_request_queue;
```

```
/* Declarations for this source file */

static struct event_str
     sample_event_queue_data[25];

class abcd_ev_queue<struct event_str>
     sample_event_queue
          (sample_event_queue_data
          , sizeof(sample_event_queue_data)
           /sizeof(sample_event_queue_data[0])
          );

static struct request_str
     sample_request_queue_data[15];

class abcd_ev_queue<struct request_str>
     sample_request_queue
          (sample_request_queue_data
          , sizeof(sample_request_queue_data)
           /sizeof(sample_request_queue_data[0])
          );

void sample_applic_task(void)
{
  int request_in_progress;
  /* ... task-specific initializations ... */

  request_in_progress=0;
  for (;;) {
```

```
    if (request_in_progress!=0) {
         abcd_wait_one_queue(&sample_event_queue);
    }
    else {
         abcd_wait_two_queues(&sample_event_queue
                             ,&sample_request_queue);
    }

    while (!sample_event_queue.is_empty()) {
      struct event_str *event_pt;
      event_pt=sample_event_queue.get_out_ptr();
      switch (event_pt->event_code) {
         case 1:
           /* ... processing for event code no. 1 ... */
           break;
         /* ... processing for other event codes .... */
         case 14:
           /* ... processing for event code no. 14 ... */
           /* Assuming event code 14 signals
                        a request completion: */
           request_in_progress=0;
           break;
      }
      sample_event_queue.advance_out();
    }
    if (   request_in_progress==0
        && !sample_request_queue.is_empty()) {
      struct request_str *request_pt;
      request_pt=sample_request_queue.get_out_ptr();
```

```
    switch (request_pt->request_code) {
       case 1:
         /* ... processing for request code no. 1 ... */
         /* Assuming request code 1 expects
                      one or more event completion: */
         request_in_progress=1;
```

```
          case 3:
            /* ... processing for request code no. 3 ... */
            /* Assuming request code 3 expects
                          one or more event completion: */
            request_in_progress=3;
            break;
          case 7:
            /* ... processing for request code no. 7 ... */
            break;
          /* ... processing for other request codes .... */
        }
        sample_request_queue.advance_out();
      }
  }
}
```

So a task that wishes to send a data packet through the radio data link simply inserts an entry in the transmission "task request queue." In practice however, various transmission requests usually fall into a few distinct urgency categories, and the above scheme uses yet another strategy to manage multiple transmit urgency categories.

Each transmit urgency category has its own "request queue," and the transmitter task selects the most urgent non-empty queue. In order to awaken the transmitter task from any one of these "request queues," a "summary request queue" is used instead of the above "task request queue." Then, a task that wishes to send a data packet inserts an entry in the relevant "request queue" (according to urgency category) and a dummy entry in the "summary request queue."

The "summary request queue" needs no entry data: it is merely a signal mechanism for the transmitter task to look at the other "request queues" in urgency order. The ABCD Proto-Kernel™ queuing mechanism supports the signal-only queues for which there is no queue entry data.

B.4.3 Residual Primarily I/O Bound Tasks

The residual primarily I/O bound tasks are at the low end of the priority range of I/O bound tasks. Slow speed serial links, low frequency external events, and data transmission activity that can tolerate gaps in the transmit data stream can fall in this category.

B.4.3.1 Miscellaneous Data Collection Task

At the low end of the priority spectrum of an ABCD Proto-Kernel™ application, it is reasonable to position the data collection functions from slow speed serial lines, or background sensor reading. In the case of slow serial lines, the intervening queues between the ISRs and this miscellaneous data collection task ensure that no data is lost and the CPU time remains dedicated to high priority tasks.

B.4.4 CPU Bound Tasks

The ABCD Proto-Kernel™ is not tuned to share the CPU resource equally between equal priority tasks in a time-sharing style. So, without precaution, a CPU bound task with can prevent the other CPU bound tasks in the group from ever doing anything. Accordingly, the scheme that governs when a task performs a computation should allow for idle times in the program sequence (e.g. between processing requests issues by other software components in the system). This does not apply to the lowest priority task which can use as much CPU time as it needs.

B.4.4.1 Route Navigation Task

The Guardian vehicle has a kind of digitized floor map of the area where it moves. The route navigation task is a potentially CPU intensive software function that computes the overall direction of the Guardian vehicle, this data being transferred to the instantaneous speed, turn, and elevation monitoring task

The route navigation task expects route computation commands on its task event queue, computes the route, and then reports completion on the caller task's event queue. From the caller task perspective, the route navigation task is just like an external peripheral that signals the completion of a request to an ISR which in turns inserts an entry in the task's event queue.

B.4.4.2 Data Synthesis Task

The data synthesis task is listed here to illustrate the possible use of background CPU processing power to analyze the Guardian collected data against the current digitized floor map, so that permanent changes in the floor plan (e.g. installation of a new furniture) are gradually recognized and integrated in the Guardian long-term memory (in the flash configuration data).

B.4.4.3 Idle task

An idle task should be provided in any ABCD Proto-Kernel™ application software. This

task should never call the **abcd_wait_one_queue** or **abcd_wait_two_queues** functions. If the idle task reserves a mutex, the other tasks that compete or this mutex must not call the **abcd_wait_one_queue** or **abcd_wait_two_queues** functions while they own the mutex. If power management functions are implemented in an ABCD Proto-Kernel™ application, the idle task might repeatedly put the system in a low power mode. Otherwise, the idle task may gather free CPU cycles statistics.