# Porting ROME to a new architecture

May 1, 2001

Leslie J. French
Distributed Systems Software Group
CCRL, NEC USA
4 Independence Way
Princeton, NJ 08540-6634

You can get the current version of this document at http://rome.sourceforge.net
For questions and comments mailto:rome-admin@lists.sourceforge.net

# Contents

# 1 Introduction

This document describes how to write a ROME system from scratch for a new architecture. In our example system the basic procedure is divided into 10 steps:

1. Gather documentation
2. Generate compiler/linker for target CPU
3. Port or write any target-specific tools
4. Design overall implementation architecture
5. Construct the target file
6. Write the cpu plugin and interrupt controller
7. Write the polled-mode I/O routimes (the uart module)
8. Load and test the *idle* process
9. Write the timer module
10. Load and execute the *perf* process (as our example application)

Each of these steps is described in its own chapter. As a concrete example, ROME will be ported to the Cyclone CVME965 development board for the Intel I960HD CPU. Brief mention will be made of the design considerations for other architectures.

# 2 Gather Documentation

Building a ROME system for an embedded application requires detailed knowledge of the hardware platform on which the system is to run. The first stage of contemplating such a project is to gather all the required information. At a minimum, this comprises the motherboard manual, processor manual and specifications for the device chips. In this case, we have:

- "Cyclone MicroSystems CVME965 Single Board Computer User's Manual" (supplied with the board)
- "I960HD User Manual" (available on the Web from developer.intel.com)
- "SCV64" (sufficient description is in the User Manual, additional data are available from www.tundra.com)
- "16550 UART" (any version of this spec. is good, for example the National Semiconductor PC16550D available on the Web).

The I960HD has an onboard timer, so there is no external timer chip needed for this system. As this is a commercial development system, it is supplied with its own boot ROM. In the case of a new hardware platform, you may have to write a boot ROM from scratch. In general, this is not straightforward and a design will be driven by many factors not directly related to ROME. Writing the boot ROM is not covered as part of this document. It is assumed that the boot ROM already exists, together with its documentation. This adds one more component to the list:

- "MON960 Debug Monitor User's Guide"

# 3 Generate Compiler

Wherever possible, ROME uses the *gcc* toolkit, which is available for a wide range of target processors. Assuming that the host environment is a 386-based Linux system, for most targets a cross-compilation environment must be created

## 3.1 Configuration

From the boot ROM manual, the target file downloaded to the system must be in COFF format, so the cross-compilation system should be built for an i960 generating COFF output. This saves an extra step in converting from a second format (such as ELF) to COFF before downloading. Apart retrieving and un-tar'ing the installation (we use the *egcs* version of *gcc*), configuring and building the *gcc* cross-compilation system is usually as easy as:

```
./configure --target=i960-coff
make
```

For full details on this procedure you should consult the gcc-cross-compiling guide.

## 3.2 Installation

Once built, the various tools must be installed into the appropriate directory within the ROME tree so that the RTB system can generate the correct paths. The Tools directory is arranged first by manufacturer (in this case Intel) and secondly by individual cpu (i960). The cross-compiler toolkit (*gcc*, *gas*, etc.) is then installed in:

```
<ROME-ROOT>/Tools/Intel/i960/bin
```

# 4 Target-Specific Tools

Although the *gcc* toolkit is quite complete, it is always possible that additional toolkit support is required. In this case, as this is the first ROME system to use COFF output, the symbol-extraction program needs to be modified. Also, an XMODEM control program is needed to download the target through the boot ROM monitor.

## 4.1 Changes to getsym.c

The *getsym* program extracts all the global symbols defined in all object files and creates the ROME symbol table as used by the debugger and the PISA environment. Unless you plan never to use the debugger or the interpreter, you will need a version of *getsym* for your environment. The initial implementation operated on ELF files, but the I960 port generated COFF files. In order to provide symbol-table support

for I960 systems, either a new version of *getsym* could be implemented, or the current version extended. Writing a new program from scratch might have been simpler, but as the output of *getsym* is closely tied to the ROME core, any changes to the symbol table format would break existing versions of *getsym*.

As it turned out, in this case, most of the logical structure of the code was unchanged when a new object-file format was added, so we were able to extend the existing program. This was possible because the different file formats can easily be distinguished. A set of COFF structures were added to represent COFF file formats, and a new procedure *dofile_coff_i960* was added to populate the symbol structure. A test was added to the *dofile* routine to check the format of each object file as it was processed and to call the appropriate handler. In this way the *getsym* program was extended, to the extent that it will cope even with mixed-format object files in a single build.

### 4.2 The xtip program

Mon960, as supplied on the CVME965 board, uses the XMODEM protocol to download (COFF) files into the processor's memory for execution, over the serial interface. This requires a terminal program on the other end of the serial line which can handle both the regular keyboard/display operation and the xmodem download. This was provided by a variant of the *tip* program, *xtip*, running on Linux, to which a download command was added. The details of this program are not relevant to the main porting issues, but the program provided the link between the cross-compilation system and the target system. In other cases this link has been bridged through the *tftp* protocol, or writing bootable floppy disks or other removable media.

## 5 General Architecture

From the assembled documentation, a number of design decisions can now be made. Although many aspects of the implementation will be fixed by the motherboard and CPU, there is still some scope for choice, and for making the right (or wrong) decisions. The general approach is to "go with the flow" and adapt the architecture to the system, rather than force a particular solution from a previous version.

### 5.1 Address Map

The I960 addressing scheme divides the address space into 16 regions, some with properties fixed by the architecture and other dynamincally configurable, for example for main RAM and VMEbus memory. Chapter 2 of the User's Manual describes the use of the front-panel switches to set the VMEbus slave address. On this particular board, the switches are set to '00100' giving a VMEbus address at 2000.0000h (from table 2-1).

Chapter 3 gives further details of the addressing. Locations 000h–800h are on-chip SRAM which can be used as a local register stack. Region a000.0000h is the main DRAM bank (which is where the program and data will be loaded) and the other memory regions are used for I/O space accesses as described in table 3-1.

Within the DRAM area, space must be allocated for system tables, in particular the Interrupt Vectors, and the stack for the interrupt handler. A good approach is to reserve stack spaces at known locations out of the way of the main code. In this case, we will start the main code partway into the DRAM space, and allocate stack above. We will set the code entry point at a00a.0000h, the supervisor stack at a005.c000h

and the interrupt stack at a005.8000h. This gives plenty of room for the interrupt stack and some space for other fixed data structures, if needed. These decisions are reflected in the contents of sections 6.3.1 and 6.5.1 below

## 5.2 Endianness

Most systems, particularly if they arrive with a boot ROM, have their endianness fixed by the architecture. In some cases, trying to force the 'wrong' endianness is almost impossible, for example PCIbus really does not work properly in a big-endian addressing regime. Usually, the only place where this is directly exposed to software is converting byte streams to integers and *vice-versa*. These conversions should properly be handled through the *ntohl* family of macros.

From table 3-2 in the User Manual, the recommended memory configuration places most of the memory in little-endian regions with the exception of the SCV64 device area in region F. Thus, the *ntohl* macros must be defined to byte-swap data in main memory.

Endiannes is also a concern when it comes to compiling and linking, to ensure that strings are compiled in the correct sequence of characters. Most *gcc/as/ld* versions allow flags to define the endiannes.

## 5.3 Caching

Handling the instruction and data caches is crucial to achieving reasonable processing on any modern CPU. The instruction cache is relatively easy, the memory region containing the executable code should be marked cacheable, and dynamic self-modifying code is *out*. Depending on the architecture, it may be necessary to perform an explicit cache flush during initialisation, for example if jump tables for interrupts are built in memory.

Handling the data cache is generally more difficult, especially for applications that expect heavy interactions with external devices. Any regions of the address space used to contain memory-mapped registers for devices must be marked uncachable. Also, any data in main memory that is read from or written by external devices must not be retained in the cache. For the CVME965 board, the private RAM area can be marked cachable, and can contain the local data and stack spaces for the ROME processes. All external DMA data will appear in the shared memory region (2000.0000h as defined above) which should therefore be marked uncachable. This leaves the implementation of the individual devices with two choices, either to access the data directly and pay the access-time penalty, or to copy into cached memory and have the data-copy overhead. The 'right' answer will depend on the data. For example data read from a disk will likely be accessed sequentially, so copying into the cache will win, whereas the first stage of filtering IP packets for addresses and ports will access only a small portion of the data; the copy can be deferred until it is known that the data will be needed.

Different approaches will apply to different architectures. In the MIPS environment, main memory is mapped into two different addressing ranges, one going first to the cache, the other bypassing the cache and accessing the memory directly. In this situation, it is possible to have close control over the caching algorithm, for example the *data* portion of all mblks is uncached, but the *control* portion is cached. For the I386 architecture, where the data cache is very large, a cache flush is an expensive operation. In this case, a small area of memory is marked as uncached as used as a buffer pool for interacting with devices, and the data are copied (exploiting the fast data-movement assembler instructions) as needed.

## 5.4 Criticality

The ROME system relies heavily on the 'critical section' to implement interlocks between processes and between contexts. There are two requirements for implementing critical sections:

1. Interrupt service routines must execute in critical sections. Also, sending a message (usually a reply) from within an interrupt service routine must *not* cause an immediate context switch, that must be deferred until the routine has completed.
2. The *rome_start_critical* routine must enter a critical section and yield some form of token indicating the previous criticality. That is, critical sections may be nested, and leaving a critical section (through *rome_end_critical*) must restore the previous state. The form of the token is not specified, to allow maximum flexibility in the implementation.

The usual approach is to disable all interrupts in a critical section, either through a 'global interrupt' flag or special machine instruction. Since this is usually a privileged (kernel or supervisor mode) operation, this alone dictates that ROME code runs in a privileged state. It is necessary that the selected method also permits a way to determine current state so that it can be restored.

Chapter 11 in the I960Hx microprocessor manual explains how the interrupt state can be altered. In this case, there are six possible ways to prevent further interrupts:

- by moving 0 to the interrupt mask register (*sf1*);
- by storing 0 in ff00.8504h, the memory-mapped location of *sf1*;
- by moving 1 to the *gie* bit in the interrupt control register (*sf3*);
- by writing 1 to the *gie* bit in ff00.8510h, the memory-mapped location of *sf3*;
- by executing the *intdis* machine instruction;
- by executing the *intctl* instruction with 0 as the first argument.

Since critical sections are used extensively within ROME, it is worth spending a little time to find the best option once the system is operating correctly. An important consideration is if the operations can be written as inline code, thereby saving the overhead of a procedure call and return. This is especially important if a procedure call is relatively expensive, for example if it causes a frame-spill on the I960. In this implementation, we will use the *intdis* and *inten* instructions to explicitly disable and enable interrupts, and the *intctl* routine to implement critical sections, since it conveniently returns the previous state.

For the I386, the only way to change interrupt state is through the assembler 'sti' and 'cli' instructions. The current state is reflected in the 0200h bit in the flags register.

## 5.5 Interrupts and Faults

Each individual architecture has its own way of handling external interrupts. For the I960, each interrupt indexes an entry of a interrupt table and causes a context switch to the address at that location. The stack pointer is set to the Interrupt Stack (unless the processor was already processing an interrupt), the *pfp*

register is set to the interrupted frame pointer, and the bottom 3 bits are set to 7, to show an interrupt-return. An interrupt record is pushed onto the stack giving the interrupt vector and the old interrupt mask. As this counts as a new procedure call, a new set of local registers are allocated for the interrupt call. The global registers retain the values they had before the interrupt. An oddity of the I960 is that each external interrupt (e.g. XINT7) is mapped to an internal interrupt vector through the *IMAP* registers which then determines the interrupt's priority. However, it is the interrupt *vector* (e.g. 0x82) that is presented to the interrupt service routine, but the interrupt *number* (e.g. 7) that must be used to clear the pending-interrupt flag.

As well as supplying a method for calling registered handlers for device interrupts, the system should also handle interrupts for which there is no driver-defined handler. At a minimum, this should report the 'unhandled interrupt' in some way, and possibly enter the system debugger.

Similarly, the system should provide a means of handling system faults. Depending on the architecture these may appear as interrupts, or there may be a separate fault system. On the I960, there is a separate table for faults (for example addressing or arithmetic exceptions). Since ROME does not provide application-level fault handlers, the usual approach is to treat faults in the same way as unhandled interrupts, and enter the debugger. The only difference is that the system supplies a fault-record on the stack instead of an interrupt vector.

## 5.6 Contexts

Every ROME process has associated with it a set of context information that must be saved when the process is suspended, and restored when the process is restarted. Typically, this will include the processor registers, the condition code and other per-process flags and the criticality. The context information must be stored under two conditions. The first is when there is an explicit contect switch due to message passing within the ROME system itelf (by a call to *cpu_suspend*). The second is when an interrupt is scheduled during a process' normal execution which causes a higher-priority process to become runnable. Often, it is possible to handle the two cases in a similar way, so that it does not matter in what way a process was suspended.

There are two basic strategies for storing the process information. The first is to use an area pointed to by the first word of the current process control block, and the second is to use the process' stack. Which option makes most sense depends on the architecture of the CPU. In the I386 case, the process stack is already used by the interrupt dispatcher to store the return information, and there are assembler routines to push and pop the register set onto the stack. In this case, the process control block field is used only to save the current stack pointer for a suspended process.

For the I960, interrupts are handled on a different stack and it is not particularly straightforward to manipulate the interrupted process' stack. In this case, the data will be stored in an area located off the current process control block. When the interrupt is scheduled, the routine is allocated a new local-register frame. This gives 12 'spare' registers (r4–r15) which can be used as temporary storage for the global registers; other data can be placed on the interrupt stack for the duration of the interrupt. Obviously this data must be preserved if the return is not to the interrupted process.

Since an interrupt causes a context switch (to the interrupt handler), processors must provide a way to restore the original context back to the executing process. Understanding how this works is a good start to design the ROME context-switcher. Often, it is possible to use this code and to combine the 'return-from-interrupt' state with the reschedule code in order to change the context. For the I960, the interupted state is indicated by the bottom 3 bits in the previous-frame-pointer register being set to '7', whereas a

'normal' return has '0' in this position. The same machine instruction (*ret*) is used on both cases. In other architectures, there are separate instructions (*ret* and *iret*) and the switching code must use the correct one. In this architecture, we can use the 'normal' return to effect a context switch even from the interrupt handler, which means that when the interrupt handler requires a context switch (instead of returning to the interrupted process), the pfp return type must be set to '0'.

When a context is restored, the criticality of the context must also be restored. Contexts may be saved in either criticality: a context switch from within the interrupt handler implies that the original process must have been *outside* a critical section (otherwise it couldn't have been interrupted); context switches while waiting for messages arise from *inside* a critical section, in this case inside *rome_await_message.* This state must be stored as part of the context information.

## 5.7 Devices

There are two, or three, basic strategies for accessing devices. They are:

1. Using special 'IOspace' instructions to move bytes or words from special locations representing devices, as found in the I386 architecture.
2. Using normal data movement instructions operating on special memory locations (memory-mapped devices).
3. Using a 'controller' to access multiple devices on a single bus (for example a SCSI controller).

The third of these access methods is not a different architecture, but requires a different style of driver. For the I960, all the devices are mapped into memory. Figure 3-1 gives the overall device layout. For example, the area at b000.0000h is used to access the registers of the onboard UART. Figure 4-4 shows the detailed memory map for this device, with one 8-bit register being decoded in each 32-bit address range. This means that the machine-dependent accesses to IOspace are simply accesses to regular memory, unlike the I386 where special machine instructions must be used. These differences are mostly hidden from drivers by defining generic macros, for example *CPU_IORD1*, to access device registers. This allows the same driver to be compiled for either memory-mapped operation or for special IOspace accesses.

Section 3.6 of the User manual describes the interrupt sources for the processor; for example XINT7 is connected to the UART interrupt. From the interrupt list, in this configuration each interrupt line is connected to a unique interrupt source, so the interrupt dispatcher can operate in 'dedicated' mode. In conjunction with section 5.5 above, the simplest approach maps XINT0 to dedicated interrupt 1 and vector 12h etc.

## 5.8 Process Initialisation

Each process in the system must be given a suitable set of initial values for the context-specific information so that it executes correctly when first started. Space must be allocated for the process' stack, and the stack frame register positioned correctly according to whether the stack increments or decrements. For the I960, the stack frame must contain the initial entry point of the process at the 'return' address and the *pfp* register must point within the stack. The process must also be started with interrupts enabled on its first context switch.

## 5.9 Optimisations

Some CPU architectures provide particular assembler assists for optimising data movement opreations. For example the I386 ‘rep stosw’ is an efficient means of clearing a block of memory to zero. The standard ROME C library provides implementations of the usual routines for these purposes (e.g. *memset*) but if faster versions are needed they can be implemented.

## 5.10 Machine-Dependent Library Support

There are two further types of routines that need to be considered as part of the machine-dependent design for the system. The first is architecture-specific support for C runtimes. Usually this means supplying the *cpu_longjmp* and *cpu_setjmp* routines, plus the data definition of the *jmp_buf* for the standard C library *longjmp* and *setjmp* calls. For the I960, these routines must preserve and restore a stack-frame environment that permits *longjmp* to execute a form of non-local-return.

The second type of routine is a true architecture-specific routine found only on this particular CPU (or family). In the case of the I960, it is necessary to be able to clear individual bits in the interrupt-pending register from within interrupt handlers. On the I386, device drivers need to read and write the machine-specific-registers (MSRs) as well as generate the IOspace instructions.

## 5.11 Design Strategy

The information gathered above must be incorporated into the design of the final ROME system. Mostly, this means either writing code, or configuring existing code through pre-processor definitions. There are five main ways of supplying configuration information:

1. by adding definitions to the ‘Hardware.h’ file section in the target file. This option should be used for definitions that are specific to the particular motherboard, for example the location of the UART registers;
2. by setting options associated with one or modules in the system. Module options should be used for configuration data that are specific to a particular project, or to a particular build of that project, for example enabling the interrupt-trace records;
3. by creating an external header file for a module. This should contain type definitions, data and procedure declarations that will need to be accessed from outside the module, for example cpu-specific routines that may be required by drivers;
4. by creating an internal header file for a module. These definitions will not be available outside the module, but provide the means to share data and definitions within a module, for example the layout of a register-set for a memory-mapped device;
5. by adding pre-processor definitions directly in a C, or assembler file. These definitions will have the most restricted scope, which is appropriate for truly local data.

As a general principle, ‘extern’ declarations should appear only in header files, not directly in source files. Except for specific standard-library examples (like *printf*) all external (non-static) definitions should contain the module prefix as the initial component. The module prefix is either the full name of the

module (e.g. *tcp*) or the class part of a compound name (e.g. *serial* in *serial_uart16550*). All variables and routines should be properly declared and type-checked. The *ptr* type should only be used when a truly opaque pointer reference is being generated or when an unknown 32-bit quantity is passed as an argument (for example in the different possible trace types).

Most of the porting effort is directed towards the central core of the ROME system. It comprises three interlinked modules:

- the machine independent 'rome' module
- the plug-in for the particular CPU in the system
- the plug-in for the interrupt controller in the system (ICU)

The functionality of the CPU and ICU modules are deliberately separated as some systems sharing the same CPU may have different interrupt controllers (for example when dispatching interrupts to PCI or VME buses). As a rule, the rome module should require *no* changes between architectures, but the other two modules will need to be written almost from scratch. Almost, because some files may be derived from existing implementations, for example many of the routines used in the debugger are machine-independent code.

The naming rules for the core of the system relax the rules slightly. In general, exports from the core use the *rome* prefix for routines provided across all architectures (e.g. *rome_start_critical*) irrespective of which actual component contains the definition. Machine-specific implementations use *cpu* or *icu* to export routines (for example to device drivers). Also, the *idle* prefix is reserved by the core to identify routines and data associated with the idle process.

## 5.12 Initialisation Flow

In order for the *ROME* module to be machine-independent, it makes certain assumptions about the structure of the cpu and icu plugins. The following diagram shows the flow of control during initialisation, and the routines that must be supplied for the *ROME* module to function:

| cpu | rome | icu | serial |
|---|---|---|---|
| *_start* | | | |
| clear BSS | | | |
| set processor state | | | |
| setup C environment | | | |
| -> | *rome_start* | | |
| *cpu_prologue* | <- | | |
| -> | | *icu_setup_def_handlers* | |
| set *cpu_freemem* | | | |
| | -> | | *serial_initp* |
| | setup memory chain | | |
| | setup trace buffer | | |
| | print copyright -> | | *serial_out* |
| | call init functions [->] | *rome_add_handler* | |
| | create process structure | | |
| *cpu_setup_process* | <- (for each process) | | |
| *cpu_epilogue* | <- | | |
| | print “Starting . . .”-> | | *serial_out* |
| *cpu_scheduler* | <- | | |
| | [*idle_process*] | | |

The ‘rome’ column of this table is fixed by the module. The table also shows the *serial* routines called to provide character-by-character output to a display during system initialisation. The calls to *rome_add_handler* come indirectly from the individual process initialisation routines. The final line represents the first context switch; in the absence of any other processes (which is *not* the normal state) this will be to the idle process.

# 6 The Target File

Once the architectural analysis described above has been followed through for a particular system, enough of the details of the motherboard and its configuration should be known that the target file can be generated. The target file describes the architecture and layout of a particular system. It allows the ROME Target Builder to generate the appropriate makefiles and the code to locate machine-specific resources.

## 6.1 Creating a new target file

Start rtb and create a new project (cvmetest) attached to your default repository. Open the project and create a new Target named cvme965LE with a suitable description (e.g. “Cyclone VME 965 board in little-endian mode”). The following definitions are entered through the various rtb dialogs. See the rtb manual for further details.

## 6.2 CPU Settings

Based on the installation directory for the tool chain chosen above, the CPU Class is “Intel” and the type is “i960”.

## 6.3 Compiler and Linker Directives

On the compiler/linker screen, select “Enable Compiler Warnings” and “Optimization 3”. The following additional flags set the assembler instruction set and listing files and the map file for the target:

```
CFlags:       -I. -fno-builtin -fpack-struct "-Wa,-al=$*.al" "-Wa,-AJX"
AsFlags:      -al=$*.al
LdFlags:      -Map=target.map
```

The current egcs version of *gcc* does not support the Hx family explicitly, so the closest architecture (JX) is used, to enable the generation of the extended instructions (notably the *inten* family).

### 6.3.1 Linker Input File

In addition to the ld flags, you must also supply the input file that *ld* will use to control the layout of the image. This file does not normally explicitly need writing for linking executables to run under an traditional operating system, as it is present in the *gcc* directory. It is needed here to control the placement of the various sections. The start of the text section, at a00a.0000h is to leave room for the supervisor and interrupt stacks as described above

```
$(OUTPUT_FORMAT(coff-Intel-little)
SECTIONS {
      .text 0xa00a0000 : { *(.text) }
      .rodata : { *(.rodata) }
      .data : { *(.data) }
      .bss : { _bssstart = . ; *(.bss) *(COMMON) ; _bssend = . ; }
}
```

This sequence also defines symbols for the start and end of the blank-storage section, so that the system initialisation code can locate and clear the area to zeroes.

## 6.4 The Make and Install Rules

The rules for converting C and assembler files to object files are mostly standard across all architectures. The ‘m’ rule converts modules written in the preprocessor for Finite State Machines and message dispatchers into standard C. The assembler is called only after the source file has been passed through *cpp* to incorporate the definitions from the target file:

```
%.c:    %.m
    state $*.m > $@

%.o:    %.s
    cpp $(INCLUDES) $*.s >/tmp/$*.t;
    $(AS) $(ASFLAGS) -o $@ /tmp/$*.t;
    rm /tmp/$*.t

%.o:    %.c
    $(CC) $(CCFLAGS) -o $@ $*.c

%.o:    %.C
    $(CXX) $(CXXFLAGS) -o $@ $*.C
```

There are no special requirements to post-process this target for installation, so no install rules are needed. For the I386 system, though, the target must be placed on a bootable floppy disk with a loader sector at the front. This is achieved with a special 'install' rule in the make file which is entered into that target's definition:

```
install:
    getprog target target.b
    (dd if=/rome/Tools/bin-
386/boot1.b bs=512 conv=sync; dd if=target.b) > boot.b
    dd if=boot.b of=/dev/fd0 bs=512 conv=sync
```

The *getprog* utility is part of the target-specific tools for the I386 environment, as is the *boot1* loader.

## 6.5 The Hardware File

The largest part of the target description is the definition of the hardware file. This is a pre-processor include file (Hardware.h) which may be used by C and assembler code to fix details of the particular system. The file generated by rtb is a combination of the hardware description defined here and the individual options selected when the system is configured. Here, the hardware file contains definitions for the initial processor state, the memory layout and the location of the devices.

### 6.5.1 Initial Processor State

The following values will be used by the assembler code as the initial values for the Arithmetic Controls, Fault Control, Interrupt Control Register-cache and Instruction-cache control registers:

```
#define CPU_INIT_AC             0x00001000
#define CPU_INIT_FC             0x40000000
#define CPU_INIT_IC             0x0
#define CPU_INIT_RC             0x0f
#define CPU_INIT_CACHE          0x00000000
```

The interrupt stack value is used to handle external interrupts, and the supervisor stack is a (temporary) stack used only during system initialisation. The size of the accessible RAM is fixed here, although the system may wish to obtain this value dynamically.

```
#define CPU_INTERRUPTSTACK      0xa0058000
#define CPU_SUPERVISTORSTACK    0xa005c000
#define CPU_RAMSIZE             (4 << 20)
#define CPU_PRIV_RAM_BASE       0xa0000000
```

In this implementation, the size of main RAM is fixed in the target file to 4M, even if the actual machine has more memory. Another approach is to define *CPU_RAMSIZE* as a variable initialised by the system at start time. The approach taken for any given system will depends on the flexibility of the CPU plugin, and the possibility of determining such values at run time. One advantage of allowing dynamic values to be overriden is that it allows a board to emulate another (perhaps development board) with less real memory.

### 6.5.2 Memory Region Definitions

As the same cpu plugin may be used with many different memory configurations, the initial values of the physical memory registers are supplied as definitions in the target file, allowing different targets to have different memory layouts without changing the code. In this case, the values are taken from the CVME965 User Guide.

```
#define CPU_REGION0             0x30800000
#define CPU_REGION1             0x30800000
. . .
#define CPU_REGIONC             0x20800000
#define CPU_REGIOND             0x20800000
#define CPU_REGIONE             0x30800000
#define CPU_REGIONF             0x30800000
```

### 6.5.3 Interrupt Map

Each (dedicated) interrupt must be mapped to a unique interrupt vector, including the special on-chip timers. These definitions allow the mappings to be set on a per-target basis.

```
#define CPU_IMAP0               0x00004321
#define CPU_IMAP1               0x00008765
#define CPU_IMAP2               0x00a90000
#define CPU_ICON                0x40c0
#define CPU_MANUAL_INTERRUPT    248
```

These are the values that map XINT0 into bit 1 and vector 12h as described in chapter 5.

### 6.5.4 Cachable Pointer Conversions

The design for the cacheing architecture uses different regions for cached and uncached memory, so there are no in-place conversions possible.

```
#define CPU_CACHED_PTR(_a)      (_a)
#define CPU_UNCACHED_PTR(_a)    (_a)
```

### 6.5.5 SCV64 Device Definitions

This section contains the CVME965 layout of the VME bus controller.

```
#define SCC_BASE                0xf0000000
```

### 6.5.6 ROME Definitions

The following definitions are used when pointer-checking is enabled in the ROME module to validate process addresses:

```
#define ROME_MIN_PPTR           (ROME_PROCESS *)0xa00a0000
#define ROME_MAX_PPTR           (ROME_PROCESS *)0xb0000000
```

### 6.5.7 Serial 16650 UART Definitions

The 16550 UART has a generic driver (see below). These definitions configure the driver for this particular motherboard in the default setting of 9600 baud.

```
#define SERIAL_UART16550_BASE0        0xb0000000
#define SERIAL_UART16550_VEC_INT0     0x82
#define SERIAL_UART16550_ADD_HANDLER  rome_add_handler
#define SERIAL_UART16550_CLEAR_INT
```

The last definition defines an 'empty' line in the source, as the interrupt does not need to be explicitly cleared.

### 6.5.8 On-chip Timer definitions

The timer uses a set of memory-mapped addresses. The system is configured for the default operation of 1,000 timer interrupts per second:

```
#define TIMER_TRR0              0xff000300
#define TIMER_TCR0              0xff000304
#define TIMER_TMR0              0xff000308
#define TIMER_INT               12
#define TIMER_VEC_INT           0x92
#define TIMER_TICKS2SEC         1000
#define CPU_FREQ_REGISTER       0xb4100000
```

Because of the vector/pin numbering scheme, both forms are defined.

### 6.5.9 Endianness Macros

The main memory is little-endian, so the *htonl* family of macros must swap bytes.

```
#define htonl(_a)               ((((uint)(_a) & 0xff) << 24) | \
                                 (((uint)(_a) & 0xff00) << 8) | \
                                 (((uint)(-a) & 0xff0000 >> 8) | \
                                 (((uint)(_a) & 0xff000000) >> 24))
#define ntohl(_a)               htonl(_a)
#define htons(_a)               (((ushort)(_a) << 8) | ((ushort)(_a) >> 8))
#define ntohs(_a)               htons(_a)
```

Most of the cpu-dependent contents of these sections can be deduced from the overall architectural decisions made above. Other definitions (for example the UART) are needed to configure pre-existing modules.

# 7 The CPU plug-in

All cpu plugins follow the same structure. They comprise the assembler routines for the initialisation of the system, and low-level operations, C routines supporting the machine-independent part of the core, and the debugger environment.

## 7.1 limits.h

The limits.h header file contains the various maximum and minimum integer values allowed for **char**, **short**, **int** and **long** variables, signed and unsigned.

## 7.2 stdargs.h

The stdargs.h header file contains the definitions for variable-count arguments. The implementation of these macros depends on how the *gcc* compiler passes arguments. This file is supplied for each architecture as part of the *gcc* distribution and the ROME file is derived from that distribution. In this case, the file is copied from *va-i960.h* with no changes.

## 7.3 stdtypes.h

The stdtypes.h header file contains definitions of extended types used throughout ROME. For most 32-bit architectures, the file can be used unchanged from one of the other distributed versions.

## 7.4 cpu_plugin.h

The cpu_plugin.h header file contains definitions for any cpu-specific structures and routines, as well as the standard routines provided as part of the ROME core.

```
typedef struct
{
    uint gregs[16];       /* saved global registers */
    uint pc;              /* saved process controls */
    uint ac;              /* saved arithmetic controls */
    uint pfp;             /* saved pfp (for ret)  */
    uint imsk;            /* save/restore imsk value */
}CPU_I960_REGISTERS;
```

This structure defines space for the 16 global registers and the process control registers. The *imsk* value is used to flag whether or not the process was inside a critical section when its context was saved.

The header file also contains the machine-dependent definition of the jump buffer used to implement *longjmp* and *setjmp:*

```
typedef struct _jmp_buf
{
    uint pfp;    /* saved pfp   */
    uint rip;    /* saved rip   */
} *jmp_buf;
```

The header file must also define the I/OSpace access macros used by devices. In this case, the macros simply reference explicit memory locations:

```
#define CPU_IORD1(_a)   *(volatile uchar *)(_a)
#define CPU_IORD2(_a)   *(volatile ushort *)(_a)
#define CPU_IORD4(_a)   *(volatile uint *)(_a)
#define CPU_IOWR1(_a, _v)  *(volatile uchar *)(_a) = (uchar)(_v)
#define CPU_IOWR2(_a, _v)  *(volatile ushort *)(_a) = (ushort)(_v)
#define CPU_IOWR4(_a, _v)  *(volatile uint *)(_a) = (uint)(_v)
#define CPU_IOSET1(_a, _v)  *(volatile uchar *)(_a) |= (uchar)(_v)
#define CPU_IOSET2(_a, _v)  *(volatile ushort *)(_a) |= (ushort)(_v)
#define CPU_IOSET4(_a, _v)  *(volatile uint *)(_a) |= (uint)(_v)
#define CPU_IOCLEAR1(_a, _v)  *(volatile uchar *)(_a) &= (uchar)~(_v)
#define CPU_IOCLEAR2(_a, _v)  *(volatile ushort *)(_a) &= (ushort)~(_v)
#define CPU_IOCLEAR4(_a, _v)  *(volatile uint *)(_a) &= (uint)~(_v)
```

These macros explicitly cast the addresses to the appropriate size, so that the same address can be used to access more than one width, and the target file can contain only the values, without needing explicit casts there.

## 7.5 _link_first.s

The assembler initialisation file is probably the most complicated part of any ROME port. The file has the special name *_link_first.s* which causes the ROME Target Builder to place it first in the constructed image. In this way, the entry point of the system is known, and fixed, at the start of the image. The file implements the following functions:

1. The initial entry point code which prepares the memory and the processor to run ROME
2. The first-level interrupt dispatcher
3. The context-switching part of the scheduler
4. Fault-handling entry to the debugger
5. Low-level cpu-dependent routines

The following code fragments do not contain the full source of the files. Rather, they focus on particular aspects relevant to the porting process.

### 7.5.1 Entry-point Code

The entry-point code contains a second entry-point at offset 4 from the main code which branches to the debugger. This can be used to restart a halted system if it traps back to the boot ROM and then use the ROME debugger to trace the fault:

```
    b       L001                # real startup code
;#
;#  Enter the debugger
;#
    ldconst  CPU_SUPERVISORSTACK,fp   # Set frame pointer
    lda      0x40(fp),sp          #  and stack pointer
    callx   _serial_initp
    callx   _rome_debug
    fmark
```

The main entry sequence is determined mainly by the “Initialisation and System Requirements” chapter (chapter 13) of the I960Hx Processor manual. First, the registers are set to good initial values:

```
L001:
    movq     0, g0                # Clear out globals
    movq     0, g4
    movq     0, g8
    movq     0, g12
    mov      0, sf0               # Clear IPND  register
    mov      0, sf1               # Mask all interrupts
    mov      0, sf2               # Clear DMA/Cache register
```

then the code clears the supervisor stack, interrupt stack and BSS area. Although clearing the stacks is not a necessary part of the operation of the system, it is only done once per reset and helps in tracing faults early in the initialisation process. The location of the stacks is picked up from the target file definition and the BSS area from the ld input script created above.

```
    lda     CPU_INTERRUPTSTACK, g0 # clear interrupt ...
    ldconst 0x8000, g1             #  ... and supv stack
    addo    g0, g1, g1
zloop0:
    stq     g4, (g0)
    addo    16, g0, g0
    cmpobl  g0, g1, zloop0
    lda     _bssstart, g0          # clear out zerovars
    lda     _bssend, g1            #  End
zloop1:
    stq     g4, (g0)
    addo    16, g0, g0
    cmpobl  g0, g1, zloop1
```

In order to move the system control tables from the boot ROM versions to those for ROME operation, the CPU must be given a software reset instruction.

```
    ldconst 0x300, r9              # r9 <- RE-INIT message
    lda     continue1, r10         # r10 <- Next IP
    lda     _ProcessControlBlock, r11   # r11 <- PRCB
    sysctl  r9, r10, r11           # Software reset
    fmark                          # If here, we are dead
continue1:
    and      r14, r14, r14         # Flush out the pipeline
    and      r14, r14, r14         # in case it is not already
    and      r14, r14, r14
```

The reset instruction loads a new process control block value which points to a data structure defined further down the file as part of the data segment. The three lines following the continuation label are no-ops to ensure the instruction pipeline operates correctly. Such additional code is defined by the architecture manual for he CPU, *and should not be omitted*. The data tables are constructed using the *CPU_XXX* values picked up from the target file hardware definitions:

```
_ProcessControlBlock:
    .word   _FaultTable            # Fault Table
```

```
        .word   _ControlTable         # Control Table
        .word   CPU_INIT_AC           # Initial AC register value
        .word   CPU_INIT_FC           # Initial Fault mask
        .word   _InterruptTable       # Interrupt Table
        .word   _SystemProcedureTable # SysProc Table
        .word   0                     # Reserved
        .word   CPU_INTERRUPTSTACK    # Interrupt Stack Pointer
        .word   CPU_INIT_CACHE        # Instr. Cache Cfg Word
        .word   CPU_INIT_RC           # Register Cache Cfg Word
```

The control table contains the interrupt map and region configuration parameters, also defined in the target file

```
    _ControlTable:
        .word   0                     # IP breakpoint register 0
        .word   0                     # IP breakpoint register 1
        .word   0                     # Data Addr bpt register 0
        .word   0                     # Data Addr bpt register 1
        .word   CPU_IMAP0             # Interrupt Map register 0
        .word   CPU_IMAP1             # Interrupt Map register 1
        .word   CPU_IMAP2             # Interrupt map register 2
        .word   CPU_ICON              # ICON register
        .word   CPU_REGION0           # Region 0 Memory Config
            . . . .
        .word   CPU_REGIONF           # Region F Memory Config
        .word   0                     # Reserved
        .word   0                     # Reserved
        .word   0                     # Trace Control Register
        .word   0x00000001            # BPCON Register
```

The interrupt table contains 256 entries, the first 9 of which are reserved. The remainder all point to the entry point of the first-level interrupt handler:

```
    _InterruptTable:
        .word     0                   # Interrupt table #1
         . . .
        .word     0                   # Interrupt table #9
        .rep    247                   # Entries #10 thru #256
        .word   _cpu_interrupt_handler   # Interrupt table entry
        .endr
```

The fault table directs all fault entries but one to the default fault handler. The exception is the ‘trace’ fault which is used to provide additional trace information for ROME processes.

```
    _FaultTable:
        .word   _cpu_def_fault_handler    # Parallel Fault
        .word   0x00000000
        .word   _cpu_trace_handler        # Trace Fault
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Operation Fault
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Arithmetic Fault
        .word   0x00000000
        .word   0x00000000                # Empty entry
```

```
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Constraint Fault
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Protection Fault
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Machine Fault
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Type  Fault
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   0x00000000                # Empty entry
        .word   0x00000000
        .word   _cpu_def_fault_handler    # Override  Fault
        .word   0x00000000
        .rep    30                        # Rest of table empty
        .word   0x00000000                # Empty entry
        .endr
```

ROME does not use system calls, so the system procedure table is empty:

```
    _SystemProcedureTable:
        .word   0x00000000          # Reserved entries
        .word   0x00000000
        .word   0x00000000
        .word   0x00000001          # Superv stack base + trace
        .word   0x00000000          # Reserved entries
        .word   0x00000000
        .word   0x00000000
        .word   0x00000000
        .word   0x00000000
        .word   0x00000000
        .word   0x00000000
        .word   0x00000000
        .rep    260                 # There are 260 entries
        .word   0x00000000          # Procedure slot
        .endr
```

Finally, the initialisation code sets up the registers to make the routine call to *rome_start* running at processor priority zero.

```
        ldconst  CPU_SUPERVISORSTACK,fp   # Set frame pointer
        lda      0x40(fp),sp               #  and stack pointer
        lda      0x001f2002, g1           # PC mask
        mov      2, g2                    # reset to priority zero
        modpc    0, g1, g2
```

```
    and      g1, g1, g1          # Wait for modpc ...
    and      g1, g1, g1
    callx   _rome_start
    fmark
```

The *fmark* instruction following the call causes the processor to fault should *rome_start* ever execute a return instruction.

### 7.5.2 Interrupt Handler

The next major component of the assembler file is the first-level interrupt handler, pointed to by all the interrupt vectors in the table above. You may be wondering why a two-level interrupt handler is necessary, it should be possible just to jump directly to the appropriate handler routine since all the interrupts come from unique sources. There are two reasons for writing the interrupt dispatcher at two levels. The first is to provide a consistent entry environment for the handlers, with a C stack, further interrupts disabled, and the ROME scheduler disabled (to prevent context switches inside interrupt handlers). The second is to handle context switches generated by messaging events inside the handler but deferred by the scheduler.

The first part of the interrupt handler saves the global registers and disables the scheduler:

```
_rome_interrupt_handler:
    intdis                      # Prevent other ext. ints
    movq    g0, r4              # Store globals in locals
    movq    g4, r8              # (can only stash 12 regs
    movq    g8, r12             #  this way)
    stt     g12,(sp)
    lda     12(sp), sp          # Carve out room on stack
    mov     0, g14              # prevent cx switches
    st      g14, _rome_allow_reschedule
```

The second part loads the interrupt vector number and call the appropriate handler:

```
#ifdef CPU_BIG_ENDIAN
    ldob    -5(fp), g0          # which intr vector (BE)
#else
    ldob    -8(fp), g0          # which intr vector (LE)
#endif
    ld       _rome_exception_handlers[g0*4], g1
    callx    (g1)               # Call the handler; ino in g0
```

Note that the byte address of the interrupt number depends on the endiannes of the machine. The code is designed to work with either endianness. In fact, if you look at the actual *_link_first.s* file, you will see that it will work across a range of I960-class CPUs, with other per-CPU tests being applied as needed. This is one reason why many of the values are set through the target file (such as the *REGION* descriptors) rather than being hard-coded into the source. You will also see that the actual code does not following this order. The reordering is described in the chapter on optimisations below.

On return from the handler, the next runnable process will be at the head of the *rome_run_queue* chain. If this is the same as the current process (*rome_this_ptr*) a context switch is not needed, and the normal return-from-interrupt sequence can be followed:

```
    mov     1, g2               # allow cx switches again
    st      g2, _rome_allow_reschedule
    ld      _rome_this_ptr, g1  # reschedule?
    ld      _rome_run_queue, g0
    cmpobne g0, g1, IRSched
    ldt    -12(sp),   g12       # Restore g12 from stack
    movq    r4,    g0           # Restore rest from locals
    movq    r8,    g4           #
    movt    r12,   g8           #
    inten                       # restore interrupts
    ret
```

Otherwise, the current interrupted process context must be saved, as though it had called *cpu_suspend*, and the system mode changed to simulate an interrupt return:

```
IRSched:
    ld      0(g1), g1           # -> register area
    stq     r8, 16(g1)          # out of the way early
    stq     r4, 0(g1)
    stq     r12, 32(g1)
    mov     pfp, r10            # r10 is PFP, old G15
    mov     1, r11              # process was enabled
    ldt     -12(sp), r4
    ldl     -16(fp), r8         # saved PC and AC -> R8,R9
    modify  7, 0, r10           # set PFP to normal return
    stt     r4, 48(g1)
    stq     r8, 64(g1)
    ldconst 0x001f2403, r9      # PC modification mask
    ld      0(g0), r4           # -> register area
    ld      64(r4), r4          # R4 PC of NEW process
    modpc   r9, r9, r4          # emulate an IRET
    b      _cpu_scheduler
```

Because this is a context-switch caused by an interrupt, the original process must have been outside a critical section, so the *imsk* flag is set to 1 (through r11) in the saved context. Since the new process may not have been saved out of an interrupt handler, all process context switches are done as ‘normal’ returns, so the return type in the *pfp* register is set to zero. However, as this is still within an interrupt handler, the ‘interrupted’ bit in the Program Controls register must be explicitly cleared, which is done by loading the PC from the new process.

### 7.5.3 Scheduler Support

The third element of the assembler file concerns the context switching mechanism. *cpu_suspend* is the routine called by the message system to save the current process’ context and switch to the next runnable process:

```
_cpu_suspend:
    ld      _rome_this_ptr, r8  # current process
    modac   0, 0, r4            # get AC value
    ld      0(r8), r8           # -> register area
    mov     pfp, r5
    mov     0, r6
```

```
    stq     g0, 0(r8)           # save globals
    stq     g4, 16(r8)
    stq     g8, 32(r8)
    stt     g12, 48(r8)
    stt     r4, 68(r8)          # AC, PFP, imask flag
```

Since *cpu_suspend* is always called from within a critical section, the *imsk* flag is set to 0, via r6. This code falls through to the *cpu_scheduler* routine, which is also called by the startup code to perform a context switch to the head of the run queue and by the interrupt handler above:

```
_cpu_scheduler:
    flushreg
    ld      _rome_run_queue, r3
    subo    1, 0, r8             # AC modification mask
    st      r3, _rome_this_ptr
    ld      0(r3), r3            # -> register area
    ldt     68(r3), r4           # R4 AC, R5 pfp, R6 imsk
    ldq     0(r3), g0
    ldq     16(r3), g4
    ldq     32(r3), g8
    mov     r5, pfp
    ldt     48(r3), g12
    modac   r8, r4, r4
    intctl  r6, r6
    ret
```

This is probably the hardest piece of code to follow in the whole of the ROME system. Firstly, any registers held in the on-chip RAM area are flushed to the stack of the current process. Then, *rome_this_ptr* is updated to point to the new process at the head of the run queue. The cpu-dependent register area addressed off the first word of the process control block is loaded and from it the global registers are restored. The condition code and other flags are restored into the Arithmetic Controls register, and the previous frame pointer is set ready for a normal procedural return. The *imsk* flag, loaded into r6, is used to control how the interrupt state is restored. If ‘1’, the process will be restored to enabled state, if ‘0’ is will (remain) disabled.

### 7.5.4 Debugger Support

The fourth element of the assembler file handles entry to the debugging system. The reason there is an assembler interface to the debugger is to provide the C code with additional information. In this case the frame pointer for the calling process’ stack is passed to the debugger as an argument:

```
_rome_debug:
    flushreg
    mov     pfp, g0           # FPointer from call stack
    callx   _cpu_idebug
    ret
```

A similar routine is used to enter the debugger on an unhandled interrupt. In this case the interrupt number is already in the first argument position, so the frame pointer is passed as the second parameter:

```
_cpu_pre_debug_int:
    flushreg
    lda     0(pfp), g1          # FPointer after int. record
    callx   _cpu_debug_int
    ret
```

A third variant of this code enters the debugger on a fault, in this case through the C *cpu_fault_routine* code:

```
_cpu_def_fault_handler:
    flushreg lda -8(fp), g0     # Load the fault record into g0
    mov pfp, g1
    modify 7, 0, g1             # Load the faulting routines fp
    callx _cpu_fault_routine
    ret
```

The assembler code is needed to extract pointers to the fault record and fault routine's frame pointer. While this *can* be done in C by manipulating the addresses of elements on the stack, it is less error-prone to pick it up from assembler.

### 7.5.5 Special Support Routines

The final part of this file contains any special assembler routines particular to this CPU. In this case, the architecture supports hardware trace events, which are handled by a special fault handler, storing the records in the standard ROME trace circular buffer:

```
_cpu_trace_handler:
    ld      _rome_trace, r4      # trace buffer
    ld      _rome_itrace, r6     # itrace variable
    ld      -8(fp), r8           # load trace type
    ld      _rome_this_ptr, r9   # current process
    stl     r8, 4(r4)[r6*16]     # save type & process
    st      rip, (r4)[r6*16]     # Address of called func.
    addo    r6, 1, r6
    ldconst 255, r5
    and     r6, r5, r6
    st      r6, _rome_itrace
    ret
```

This section also contains the support routines for the C library *longjmp* and *setjmp* routines. A longjump is implemented as a procedure return to a different return address from the one called, and with a different stack frame.

```
_cpu_setjmp:
    flushreg   # no cached registers
    st pfp, 0(g0)  # save pfp
    ld 8(pfp), r4
    st r4, 4(g0)  # save RIP
    mov 0, g0
    ret
_cpu_longjmp:
```

```
    flushreg   # cached is unsafe
    ld 0(g0), pfp   # restore pfp
    ld 4(g0), r4
    st r4, 8(pfp)  # restore RIP
    mov g1, g0   # return code
    ret
```

Because the call chain is broken in *longjump* the register cache must be flushed. Similarly, the cache must also be flushed in *setjmp* to ensure that the RIP is in the stack frame. The *longjump* code effectively moves the stack back to the point when *setjmp* was called, and returns to the point from which *setjmp* was called, but with the return code supplied to *longjmp*.

This completes the contents of the *_link_first.s* file.

## 7.6 k960.c

The 'kernel' file contains the C routines used to complete initialisation of the system. First comes some data definitions:

```
ROME_TRACE *rome_trace;    /* ROME event tracing array  */
int rome_itrace = 0;       /* current trace index       */
uint cpu_freemem;          /* start of free memory      */
```

The *rome_trace* array is shared with the assembler trace fault handler to provide a circular buffer of trace events. The API to the trace system is through the *rome_add_trace* routine:

```
void rome_add_trace(ptr a0, int a1, ptr a2)
{
    rome_trace[rome_itrace].address = a0;
    rome_trace[rome_itrace].type = a1;
    rome_trace[rome_itrace].current = rome_this_ptr;
    rome_trace[rome_itrace].spare = a2;
#ifdef CPU_KPRINTF_TRACES
    rome_kprintf("@%x proc %x code %02x arg %08x\n", a0,
                 rome_this_ptr, a1, a2);
#endif
    rome_itrace = (rome_itrace + 1) & 255;
}
```

The CPU_KPRINTF_TRACES option prints trace requests as they are generated from explicit calls within the code (but not from the trace-fault handler). This is a useful debugging tool for early system debugging.

The machine-independent core calls the *cpu_prologue* routine to perform any early cpu-specific initialisation. One requirement is that the *cpu_freemem* variable must be initialised on return from this routine. Because this routine is called very early in the system, no memory may be allocated here, and the *rome_kprintf* routine is not available. The *prologue* routine completes the initialisation of the memory manager by enabling the data cache on main memory, and setting the SVC64 region big-endian, as described in the User Manual.

```
void cpu_prologue(void)
{
    extern int bssend;
    /* default uncached, little-endian */
    CPU_IOWR4(CPU_DLMCON, 0);
    /* region A, cached */
    CPU_IOWR4(CPU_LMAR0, 0xa0000002);
    CPU_IOWR4(CPU_LMMR0, 0xf0000001);
    /* region F uncached, big endian */
    CPU_IOWR4(CPU_LMAR14, 0xf0000001);
    CPU_IOWR4(CPU_LMMR14, 0xf0000001);
    icu_setup_default_handlers();
    cpu_freemem = &bssend;
}
```

The *cpu_epilogue* routine is called after all the processes have been initialised, just before the scheduler is called for the first time:

```
void cpu_epilogue(void)
{
    int i;
    rome_this_ptr = (ROME_PROCESS *)&i;
}
```

The *cpu_setup_process* routine is called once for each main process in a module. It is not called for processes that only have initialisation routines and do not then receive messages (‘initonly’ processes). The routine must perform all the machine-dependent initialisation of the process structure. In this case, that means allocating and setting up the stack and the process’ register area:

```
void cpu_setup_process(ROME_PROCESS *here, ROME_INIT_PROC *proc)
{
    CPU_I960_REGISTERS *regs = (CPU_I960_REGISTERS*)
                               rome_alloc(sizeof(CPU_I960_REGISTERS), 3, 1);
    here->rome_stack = (int *)rome_alloc(proc->stksize, 2, 0);
    here->rome_stack[1] = ((uint)here->rome_stack)+64;
    here->cpu_dep = regs;
```

The process entry point is either to the process’ main routine, or to *_main* to initialise the standard I/O files:

```
    if  (proc->main_flag)
    {
#ifdef ROME_NO_STDIO
        rome_fatal("Stdio process in this system!");
#else
        here->rome_stack[2] = (int)_main;
        regs->gregs[0] = (int)proc->main;
        regs->gregs[1] = (int)proc->name;
#endif
    }
    else
    {
        here->rome_stack[2] = (int)proc->main;
    }
```

The return frame pointer is set to the frame containing the start address, and the rest of the registers are set from the process' initialisation record:

```
    here->rome_stack[0] = (int)here->rome_stack;
    regs->gregs[15] = (int)here->rome_stack + 64;
    regs->ac = 0x00001000;
    regs->pc = (2 | proc->trace_flag);
    regs->pfp = (int)here->rome_stack;
    regs->imsk = 1;                      /*  restore to global */
}
```

The routine also checks if the project has disabled standard-I/O processing, and if so fatals if any process is declared as requiring stdio. This is mainly for extremely small ROME systems that are using a minimal C library.

## 7.7 debug.c

Most of the debugger is common code, for example to display memory or to format messages. The machine-dependent part of the debugger is concerned with printing machine registers and formatting the trace output. First comes a list of the machine-specific faults and traces:

```
#define TRACE              0x01        /* Trace           */
#define OPERATION          0x02        /* Operation       */
#define ARITHMETIC         0x03        /* Arithmetic      */
#define CONSTRAINT         0x05        /* Constraint      */
#define PROTECTION         0x07        /* Protection      */
#define TYPE               0x0A        /* Type            */
/* Fault Sub-Types */
#define ITRACE             0x02        /* Instruction Trace */
#define BRTRACE            0x04        /* Branch Trace    */
#define CTRACE             0x08        /* Call Trace      */
#define RTRACE             0x10        /* Return Trace    */
#define PTRACE             0x20        /* Prereturn Trace */
#define STRACE             0x40        /* Supervisor Trace*/
#define BPTRACE            0x80        /* Breakpoint Trace*/
#define INVOPCODE          0x01        /* Invalid Opcode  */
#define UNIMP              0x02        /* Unimplemented   */
#define UNALIGN            0x03        /* Unaligned       */
#define INVOPERAND         0x04        /* Invalid Operand */
#define INTOVER            0x01        /* Integer Overflow*/
#define DIVZERO            0x02        /* Divide by Zero  */
#define CRANGE             0x01        /* Constraint Range*/
#define PRIV               0x02        /* Privileged      */
#define LENGTH             0x01        /* Length          */
#define TYPEMIS            0x01        /* Type Mismatch   */
```

The *print_cpu_registers* routine is called from the process display code to format the machine-specific part of the process structure. This prints the global registers and the top of the stack:

```
static void print_cpu_registers()
{
    uint *proc_fp;
```

```
    rome_kprintf("G0:  %x %x %x %x\n", regs->gregs[0], regs->gregs[1],
                 regs->gregs[2], regs->gregs[3]);
    rome_kprintf("G4:  %x %x %x %x\n", regs->gregs[4], regs->gregs[5],
                 regs->gregs[6], regs->gregs[7]);
    rome_kprintf("G8:  %x %x %x %x\n", regs->gregs[8], regs->gregs[9],
                 regs->gregs[10], regs->gregs[11]);
    rome_kprintf("G12: %x %x %x %x\n", regs->gregs[12], regs->gregs[13],
                 regs->gregs[14], regs->gregs[15]);
    rome_kprintf("PC: %x  AC:  %x  PFP: %x IMSK% %x\n", regs->pc,
                 regs->ac, regs->pfp, regs->imsk);
    proc_fp = (uint *)(regs->gregs[15] & 0xfffffff0);
    rome_kprintf("Stack-- fp: %x  ", proc_fp);
    rome_kprintf("pfp: %x  ", proc_fp[0]);
    rome_kprintf("sp: %x  ", proc_fp[1]);
    rome_kprintf("rip: %x\n", proc_fp[2]);
}
```

The *traceback* routine gives a call-by-call traceback of a process' stack, where possible:

```
static void traceback(char *args)
{
    uint *cfp = (uint *)((uint)fp & (~15));
    int i = 0;
    int res;
    char *tmp;
    rome_kprintf("Starting traceback at %x\n", cfp);
    while (cfp > (uint *)CPU_PRIV_RAM_BASE && i < 20)
    {
        rome_kprintf("pfp %x sp %x rip %x ", cfp[0], cfp[1], cfp[2]);

        if  ((tmp = find_symbol(cfp[2], &res)) == NULL)
        {
            rome_kprintf("\n");
        }
        else if  (res == 0)
        {
            rome_kprintf(" = %s\n", tmp);
        }
        else
        {
            rome_kprintf(" = %s+%x\n", tmp, res);
        }
        i++;
        cfp = (uint *)(cfp[0] & (~15));
    }
}
```

The *traceback* routine relies on the *pfp* register linkage to move to the caller's stack frame. On architectures that do not support such linked frames, where each procedure 'knows' how much stack is allocated, this may be impossible to implement.

The *display_local_reg* routine displays the local registers on the stack of the current process:

```
static void display_local_reg(char *args)
{
```

```
    uint *ptr = fp;
    rome_kprintf("frame pointer: %x\n", fp);
    rome_kprintf("pfp:  %x   sp:  %x  rip:  %x   r3:  %x\n",
                 ptr[0], ptr[1], ptr[2], ptr[3]);
    rome_kprintf(" r4:  %x   r5:  %x   r6:  %x   r7:  %x\n",
                 ptr[4], ptr[5], ptr[6], ptr[7]);
    rome_kprintf(" r8:  %x   r9:  %x  r10:  %x  r11:  %x\n",
                 ptr[8], ptr[9], ptr[10], ptr[11]);
    rome_kprintf("r12:  %x  r13:  %x  r14:  %x  r15:  %x\n",
                 ptr[12], ptr[13], ptr[14], ptr[15]);
}
```

On the I960, this routine relies on the *flushreg* instruction in the various fault handler support routines to move all the registers from the on-chip SRAM into the main stack.

The *trace_it* process formats the trace records strored in the trace structure. This is a long routine, so a sample only is given here:

```
static void trace_it(char *args)
{
    ROME_TRACE *ptr;
    int res;
    int i;
    int j = 256;
    i = rome_itrace;
    while (j--)
    {
        char *rn;
        ptr = &rome_trace[i];
        if  (ptr->type & 255)
        {
            rome_kprintf("%s:  ", ptr->current->name);
            switch (ptr->type & 255)
            {
              case ITRACE:
                rome_kprintf("Itrace at %x", ptr->address);
                break;
                . . . .
              case ROME_TT_STARTINT:
                rome_kprintf("Start interrupt %x at %x", ptr->spare, ptr-
>address);
                break;
                . . .
              default:
                rome_kprintf("Event %d at %x", (ptr->type & 255), ptr-
>address);
                break;
            }
```

The switch statement must handle both the architecture-specific trace entries (such as *ITRACE*) and the generic entries common to all implementations, such as *ROME_TT_STARTINT.* The routine also tries to make use of the symbol table to interpret the address field in the record:

```
            if  ((rn = find_symbol(ptr->address, &res)) != NULL)
```

```
            {
                if  (res == 0)
                {
                    rome_kprintf(" (%s)", rn);
                }
                else
                {
                    rome_kprintf(" (%s+%x)", rn, res);
                }
            }
            rome_kprintf("\n");
        }
        i = (i+1) & (ROME_TRACE_COUNT-1);
    }
    return;
}
```

The debug.c file also contains the code for the unhandled interrupt handler. The first part of the code does a sanity check that the interrupt vector and interrupt stack record are consistent:

```
void cpu_debug_int(int vector, uint *frame)
{
    if  (vector != (frame[-2] & 0x000000ff))
    {
        rome_kprintf("\nVector (%x) and record (%x) do not match\n",
                     vector, frame[-2]);
        rome_kprintf("Interrupt Stack may be invalid \n");
    }
```

it then prints out the unhandled interrupt frame:

```
    rome_kprintf("Frame at %x\n", frame);
    rome_kprintf("\n  Vector Number (from interrupt record): %x\n",
                 frame[-2] & 0x000000ff);
    rome_kprintf("  IP: %x\n", frame[2]);
    rome_kprintf("  PC: %x\n", frame[-4]);
    rome_kprintf("  AC: %x\n", frame[-3]);
    rome_kprintf("  FP: %x\n", frame);
    rome_kprintf("Note: The interrupt stack is the current stack.\n");
```

The remainder of this routine handles the entry to the debugger:

```
    if  (vector == I_MANUAL)
    {
        rome_kprintf("Non-maskable interrupt occurred.\n");
        rome_idebug(frame);
    }
    else
    {
        rome_kprintf("Unhandled interrupt has occurred.\n");
        . . .
     }
}
```

The fault handler is called whenever a machine fault (as opposed to an interrupt) is detected. This routine is called from the assembler routine described above and prints out the fault information and calls the debugger.

```
void cpu_fault_routine(uint *fault_record, uint *fp)
{
    int f_type;
    int fs_type;
    int old = rome_start_critical();
    f_type = ((*fault_record >> 16) & 0x000000ff);
    fs_type = (*fault_record & 0x000000ff);
    rome_kprintf("****FAULT RECORD at %x: ", fault_record);
    rome_kprintf("address %x, type %x = ", fault_record[1], fault_record[0]);
    switch(f_type)
    {
    case OPERATION:
        rome_kprintf("Operation-");
        switch(fs_type)
        {
        case INVOPCODE:
            rome_kprintf("Invalid Opcode\n");
            break;
           . . . .
        }
        break;
    . . . .
    default:
        rome_kprintf("Unknown Fault Type %d.\n", f_type);
        break;
    }
    rome_kprintf("Entering the debugger with frame %x\n", fp);
    cpu_idebug(fp);
    rome_end_critical(old);
}
```

These are are the only machine-dependent routines in the debugger.

## 7.8 disassembler.c

The disassembler routine formats memory as instructions when called from the debugger *dis* command or directly from within application code. The routine is prototyped as:

```
void cpu_disassembler(
        uint **instr)
```

It should format the data at address **instr* as a machine instruction (if possible, else print out a 32bit hex number), and update the *instr* pointer to point beyond the instruction (for machines with variable-length instructions, such as the I386). All output should be done through *rome_kprintf*. There is little point in describing how to write a disassembler here, I'm sure you can find one or adapt one of the supplied versions to suit your needs.

# 8 The ICU plug-in

The ICU (Interrupt Control Unit) plugin is the component of the ROME core that dispatches interrupts to device drivers.

## 8.1 icu.h

The icu header file contains the standard exported data and routines common to all interrupt controllers, plus any local additions. For the I960, four additional routines are defined, to enable and disable individual interrupts and to handle the pending interrupt flags. *icu.h* also prototypes the three standard rome routines *rome_add_handler*, *rome_end_critical* and *rome_start_critical*. The other routines in the module are machine-dependent and use the *icu* prefix.

## 8.2 icu.c

The main interrupt-control unit code contains the storage definition for the handlers vector, exported through the header file:

```
int icu_exception_handlers[260];       /* shared with cpu plugin         */
```

The routines to enable and disable interrupts set or clear the appropriate bits in the memory-mapped *IMSK* register:

```
void icu_enable_interrupt(int ino)
{
    CPU_IOSET4(CPU_IMASK_ADDR, BIT(ino));
}
void icu_disable_interrupt(int ino)
{
    CPU_IOCLEAR4(CPU_IMASK_ADDR, BIT(ino));
}
```

The *rome_add_handler* routine adds a routine pointer into the table and enables the coresponding interrupt. Because the interrupts are configured in dedicated mode, XINT0 corresponds to vector 12h etc. so the mask bit can be calculated from the vector number:

```
void rome_add_handler(int where, void (*rtn)(int))
{
    int bitno = (where >> 4) - 1;
    icu_exception_handlers[where] = rtn;
    icu_enable_interrupt(bitno <= 7 ? bitno : bitno+4);
}
```

The test in the call to *icu_enable_interrupt* handles the case of the timer interrupt which occupy bit positions 12 and 13, but generate vectors 92h and a2h.

The *icu_setup_default_handlers* routine is called from the *cpu_prologue* to initialise the interrupt handlers to a known state before any of the driver initialisation routines are called (which may modify the table by calling *rome_add_handler*)

```
void icu_setup_default_handlers()
{
    int i;
    for (i = 0; i < 256; i++)
    {
        icu_exception_handlers[i] = (int)cpu_pre_debug_int;
    }
}
```

### 8.3 icu_asm.s

The assembler portion of the ICU module contains routines that cannot easily be expressed in C. This includes the two routines for critical section support:

```
_rome_start_critical:
    intctl 0, g0
    ret
_rome_end_critical:
    intctl g0, g0
    ret
```

The *start* routine uses the *intctl* instruction to disable interrupts while returning the previous interrupt state into g0 (and so back to the caller). The *end* routine uses the supplied value in g0 to restore the state to that level of criticality.

The file also contains two routines for examining and clearing the pending-interrupt flags:

```
_icu_clear_ipend:
    ldconst CPU_PRIV_RAM_BASE,r5
    atmod r5, 0, r5     # Clear out the bus unit
    ldconst CPU_IPND_ADDR,r5   # r5 -> IPND register
    movl 0, r6          # Mask and data registers
    setbit g0, r6, r6   # Set bit in mask register
    atmod r5, r6, r7    # Do it
    ret
_icu_check_ipend:
    ld CPU_IPND_ADDR, r6 # get IPND contents
    bbc g0, r6, clr      # check and branch if bit is clear
    mov 1, g0            # bit not clear return 1
    ret
clr: mov 0, g0
    ret
```

The *clear* routine is slightly complicated because of the need to ensure that clearing a bit does not interfere with any other bits being set asynchronously from external sources, hence the use of the atomic-modify operations.

## 9 The Serial Interface

At a minimum, the system needs some basic character-mode input and output, particularly during initial development. Usually, this is provided through a UART connected to a serial line. In this case, the board

has an built-in TI16C550 UART chip. As this is a very common chipset, there is already a ROME driver for this UART, in the *serial_uart16550* module. The module was written to be quite generic. From the module documentation, a number of parameters need to be configured to use the driver. These are:

| name | description | default | T/O |
|---|---|---|---|
| SERIAL_UART16550_ADD_INCLUDE | additional include file | *notdef* | T |
| SERIAL_UART16550_REG_SPACING | addressing step | 4 | T |
| SERIAL_UART16550_BAUD_SHIFT | baud rate multiplier | 4 | T |
| SERIAL_UART16550_BAUD_RATE | line rate | 9600 | T |
| SERIAL_UART16550_XTAL_FREQ | crystal frequency | 1843200 | T |
| SERIAL_UART16550_BASE0 | base of registers | none | T |
| SERIAL_UART16550_NO_POLLEDIO | don't define polled routines | *unset* | O |
| SERIAL_UART16550_SKIP_INIT | do not process init | *unset* | O |
| SERIAL_UART16550_ENTER_DEBUG | chracter to trigger debugger | *unset* | O |
| SERIAL_UART16550_CLEAR_INT0 | code to clear interrupt | *notdef* | T |
| SERIAL_UART16550_VEC_INT0 | interrupt vector number | none | T |
| SERIAL_UART16550_ADD_HANDLER | routine to add handler | none | T |

The final column indicates a value in the Target file or a module Option. The three values labelled 'none' in the table must be supplied in order that the module will build. The others need only be defined or changed if required. From the motherboard manual the base address and register spacing of the UART are known, and these values have already been entered into the target file above.

If you are writing your own serial interface driver you will need to define your own set of values. All serial modules should support the three options: *NO_POLLEDIO*, to stop generation of the *serial_poll* etc. routines (in the case where a system has multiple serial interfaces); *SKIP_INIT*, to disable the actions of *serial_initp* during system debugging; and *ENTER_DEBUG*, which should be set to a character constant to cause entry to the debugger from the serial interrupt handler. The module should otherwise provide the four polled-mode interface routines: *serial_in, serial_initp, serial_out* and *serial_poll* in addition to handling all the messages from the *Standard* message-set.

It is also a good idea to provide a routine that can be called from the debugger to display the state of the serial interface.

# 10 Testing the idle process

At this point enough of the system is in place to build and test it. Using RTB, load the cvmetest project that you used to enter the target file above and create new modules for CPU_I960, of class Kernel.PlugIn and ICU_I960, of class Driver.Interrupt. Add the source and header files to the modules and mark the header files as exported. Use CVS to check out the four remaining modules for a minimal system: ROME; rome_if; Clib; and serial_uart16550. If you have had to write your own serial driver, then enter that module into the project and use it instead of the serial_uart16550 one. Also check out the *Standard* message-set and mark it in-use. Build the system from within RTB and then 'make' it in the Modules directory. If all goes well, you should get a target file, and a target.map file (and no warnings on the compilations). If you then download the system onto your target board and start it, you should see something like this:

```
ROME Initialising.
```

```
Copyright 1997 NEC USA Inc.
Built by leslie on pc-rome.ccrl.nj.nec.com at Thu Jan 11 17:30:06 2001
Starting the Scheduler.....
```

Followed by nothing. If you do, congratulations, you've got quite a long way already. If you don't, well, my systems don't usually work the first time either, so skip to the "What if it doesn't work?" section below.

If you got this far, then go back to the project view in RTB and select the options screen. Enable the option named *SERIAL_UART16550_ENTER_DEBUG* and rebuild the system. Now, when the system is running, the '!' key should drop you into the debugger, like this:

```
!
rome debug>
```

Now you can try out some of the debugger commands, and make sure the system is really there, for example you can list the (two) processes in the system:

```
rome debug> lp
   Process serial located at 0xa03fede0.
   Process idle located at 0xa03fdbd0.
Process pointer is currently pointing to idle.
```

Try disassembling the code at the entrypoint and comparing it to the *_link_first.s* file:

```
rome debug> dis a00a0000
0xa00a0000 0x08000024            b 0x00000024
0xa00a0004 0x8cf83000 0xa005c000 lda 0xa005c000, fp
0xa00a000c 0x8c0fe040            lda 0x00000040(fp), sp
0xa00a0010 0x86003000 0xa00aa0a0 callx 0xa00aa0a0
0xa00a0018 0x86003000 0xa00a0158 callx 0xa00a0158
0xa00a0020 0x66003e00            fmark 0xa00a0024
0x5f801e00                       movq 0, g0
0xa00a0028 0x5fa01e00            movq 0, g4
0xa00a002c 0x5fc01e00            movq 0, g8
```

Then check the register values in one of the processes:

```
rome debug> cp serial
Switching process pointer to process serial.
rome debug> pinfo
Process serial at 0xa03fede0.
G0: 0x00000000 0x00000000 0x00000000 0x00000000
G4: 0xa03fede0 0x00000002 0xa03fdbd0 0x00000000
G8: 0x00000000 0x00000000 0x00000000 0x00000000
G12: 0x00000000 0x00000000 0x00000000 0xa03fddb0
PC: 0x00000002 AC: 0x00001002 PFP: 0xa03fddb0 IMSK 0x00000000
Stack-- fp: 0xa03fddb0 pfp: 0xa03fdd70 sp: 0xa03fddf0 rip: 0xa00a96f0
State: 2, Priority: 5
No messages on queue.
```

You can go on to look at the stack, other memory, tracebacks etc., but by now, this is a good indication that your debugger is working properly (you're sure to need it later) and that there really is a ROME system running on this machine.

# 11 The Timer

The second essential component for most system is a timer process, to generate timeouts at programmable intervals for applications. The I960Hx processor has an on-chip timer accessed through memory-mapped registers. The addresses for these registers have already been entered into the target file above. Since the clock frequency is determined by the bus clock rate, it is necessary to know the processor speed in order to detemine the clock counter interval. The CVME965 board has another memory-mapped location which gives an index value into a table of CPU frequencies. This is the *CPU_FREQ_REGISTER* location defined in the target file. The usual configuration for the timer generates one interrupt every millisecond, giving a *HZ* value of 1,000. Under some circumstances (see below, and the next section) it is interesting to run different clock values. The standard *TIMER_DEBUG* option allows the default values to be overriden with inlined values in the source, and to print the idle loop counter after a number of ticks have elapsed.

## 11.1 timer.c

This is the main driver source file. As with the other files, only extracts of the code will be used to describe particular points. There is a ‘standard’ means of implementing ROME timer queues, where the *TIMER* messages are linked in a single chain in expiration order, with the *ticks* values giving the number of ticks *between* each message on the queue. In this scheme, only the top message of the queue is examined on each tick, and all messages at the head of the queue with a *ticks* value of zero expire at the same time. This slows down the addition of messages to the queue, since the delta values must be subtracted for each message, but makes the interrupt code faster. Of course, you can do it a different way (and tell the rest of us if it’s better), but this example uses the ‘standard’ code.

Another ‘standard’ fearture of the timer module is the *TIMER_DEBUG* option which overrides the standard values as follows:

```
#ifdef TIMER_DEBUG
#define TIMER_DEBUG_TIX 3000
#define TIMER_DEBUG_CNT 300000
static int tick_ending = TIMER_DEBUG_CNT + 1;
static uint cit1, cit2;
#endif
```

In normal operation the routine counts seconds and sub-seconds:

```
static int secs = 0;
static int tick_counter;
static ulong tc;
```

The timer initialisation routine programs the values into the timer to interrupt the system *TIMER_TICKS2SEC* times a second:

```
void timer_init(void)
{
    tick_counter = 0;
    timerq = NULL;
    rome_add_handler(TIMER_VEC_INT, timer_isr);
#ifdef TIMER_DEBUG
```

```
    tc = clock_table[CPU_FREQ] / TIMER_DEBUG_TIX;
#else
    tc = clock_table[CPU_FREQ] / TIMER_TICKS2SEC;
#endif
    CPU_IOWR4(TIMER_TMR0, 0);
    CPU_IOWR4(TIMER_TRR0, tc);
    CPU_IOWR4(TIMER_TMR0, TMR_AUTO_RELOAD | TMR_SUP_WRITE | TMR_1_X_CLOCK);
    icu_clear_ipend(TIMER_INT);
    CPU_IOSET4(TIMER_TMR0, TMR_ENABLE);
}
```

The *tc* variable is the number of bus clock cycles in each timer interval. The timer is programmed for continuous operation with automatic reloading of the counter when it expires.

Messages are added to the timer queue from the queue handler, so no context switch is needed into the main process. For the default linking strategy, the timer queue handler routine is common to all implementations and is not described in detail here. The routine first returns any messages with zero or negative time. Otherwise the message is linked into the timer queue in expiration order. If the timer queue is empty, this becomes the only message on the queue. Note that as the queue handler runs as a critical section, the queue manipulation is automatically protected from the asynchronous accesses made by the interrupt routine.

The main process itself does nothing, it exists only to give a process control block with the name “timer” for the shared library.

```
void timer_process(void)
{
    while (1==1)
    {
        ROME_MESSAGE *m = rome_await_message(0, 0);
        rome_kprintf("timer message %x??\n", m);
    }
}
```

This code will only be invoked in the (unlikely) event that a message other than *TIMEOUT* is sent to the timer process.

The interrupt handler then operates only on the top element of the queue:

```
void timer_isr(int ino)
{
    if (timerq)
    {
        register ROME_T_TIMEOUT *tp = RCASTP(ROME_T_TIMEOUT, timerq);
        tp->ticks--;
        while (timerq && tp->ticks <= 0)
        {
            ROME_MESSAGE *head = timerq;
            timerq = head->link;
            rome_reply(head);
            tp = RCASTP(ROME_T_TIMEOUT, timerq);
        }
    }
```

The handler also maintains the tick counter, and clears the interrupt-pending bit:

```
        if (++tick_counter == TIMER_TICKS2SEC)
        {
            secs++;
            tick_counter = 0;
        }
        icu_clear_ipend(TIMER_INT); /* Clear the IPEND bit */
    }
```

The operation of the handler may be modified by the *TIMER_DEBUG* flag. In this mode, the handler first stores the values of the idle process' loop count, then after a programmable number of ticks, displays the number of idle loops elapsed:

```
        if (--tick_ending == TIMER_DEBUG_CNT)
        {
            cit1 = idle_one;
            cit2 = idle_two;
        }
        else if (tick_ending == 0)
        {
            uint dt1, dt2;
            if (cit1 > idle_one)
            {
                dt1 = 0xffffffff - cit1 + 1 + idle_one;
                dt2 = idle_two - (cit2+1);
            }
            else
            {
                dt1 = idle_one - cit1;
                dt2 = idle_two - cit2;
            }
            rome_kprintf("%d %d\n", dt2, dt1);
        }
```

The computation is designed to handle the case where the machine idles so rapidly that more that $2^{32}$ idle loops occur in the interval.

Finally, the timer module contains a routine that can be called from the debugger to dump the contents of the timer queue. In this manner, the debugger is largely independent of the particular arrangement or implementation details of the individual modules, and the programmer can still get useful information in the event of a crash or strange operation:

```
    void timer_show_timerq()
    {
        ROME_MESSAGE *head = timerq;
        while (head)
        {
            register ROME_T_TIMEOUT *tp = RCASTP(ROME_T_TIMEOUT, head);
            rome_kprintf("R %d -> %s\n", tp->ticks, head->src->name);
            head = head->link;
        }
    }
```

The other routine in the source file, *timer_tag* returns the current timer count values for programs wishing to do their own interval timings.

### 11.2 timerlib.c and timerlib.h

The *timerlib.c* and *timerlib.h* files contain the machine-independent interface to the timer, handling the message queueing and callback routines. These two files can be copied from any existing timer module into your own module and should not require any modifications.

### 11.3 Timer Testing

Add the new timer module into your project, and rebuild the system. The system should run as before, except that the "timer" process should be in the system. To verify the basic interval timer, enable the *TIMER_DEBUG* option and modify the source with the following values:

```
#define TIMER_DEBUG_TIX 1000
#define TIMER_DEBUG_CNT 120000
```

When you next start the system, use an external timer (stopwatch or another computer) to measure the time interval between starting the scheduler and the timer code printing the idle counts. This should be almost exactly 2 minutes. This code tests that the timer values have been programmed correctly, and that the interrupt handler is capable of processing the 120,000 interrupt needed to get this far. Longer-running timing tests using *Perf* below can be used to tune the exact value for the timer registers, at this stage it is sufficient for it to be within a second or two of the real value.

## 12 Interrupt Timing

The timing debug mode can be used to estimate the processing time required to handle the timer interrupt. Given that the timer is a very simple interrupt handler, this gives a good idea of the basic interrupt dispatching overhead in the system. The idea is to note the consumption of extra idle loops as more and more timer interrupts are processed per second. Run the timer debug code with the *TIX* value set to 1000, 2000 .. 5000 and the *CNT* value set to $100 \times TIX$ (i.e. to make the measurements over 100 seconds independently of the *TIX* value). This should produce a set of results with the following trend (but obviously with different numbers for a different motherboard):

| ticks/sec | idles/sec | delta |
|---|---|---|
| 1,000 | 2.750,740 | |
| 2,000 | 2,741,674 | 9,066 |
| 3,000 | 2,732,415 | 9,259 |
| 4,000 | 2,723,391 | 9,024 |
| 5,000 | 2,714,362 | 9,029 |

This shows that each interrupt is equivalent to 9 idle loops, and that an uninterrupted system would run 2,759,800 idle loops/second, which is equivalent to 306,600 timer interrupts. Thus each timer interrupt takes approximately 3.25 $\mu$s. This calculation is very approximate, but it gives a 'ballpark' figure for working out how the system will perform under real load.

# 13 Running the perf process

By now you are in a position to make some useful performance measurements on your system in addition to the interrupt handling time measured above. In the RTB project, unset the *TIMER_DEBUG* option, and add to the project the *Console* module and the *Perf* module and rebuild. *Perf* measures the peformance of the basic rome messaging core by sending and replying to messages in the various configurations possible, testing both Queue Handling and full Context switching. Running *Perf* produces four numbers in a continuous loop, with one measurement every 10 seconds. The number are:

Queue Handling The time taken to send a single message which is replied to from within a queue handler and to receive the reply at the sending process.

Context Switching The time taken to pass a message through a Queue Handler (unhandled) into a main process, generate the reply (not passing through a Queue Handler) and switch back to the sending process to receive the reply. That is, this number represents 1 Queue-Handler plus 2 individual context switches.

Routine Call The time to call a routine, which just increments a counter and returns.

Protected Routine Call The time to call a routine which starts a critical section, increments a counter, ends the critical section, and returns.

The time for a individual context switch is $(CS - QH)/2$. Running *Perf* on the system as built above gives the following numbers:

| field | value ($\mu$s) |
|---|---|
| Queue Handling | 6.93 |
| Context Switch | 37.10 |
| Routine Call | 0.41 |
| Protected Routine Call | 0.99 |
| Single Context Switch | 15.09 |

These are not bad numbers for a 33MHz system. These numbers serve as a useful comparison against other operating systems, as long as the other figures are measuring the same values. The *Perf* values represent real user–user messaging, not the 'kernel-only' part or some carefully-crafted subset of the real functionality. The interrupt dispatching value measured above (3.25 $\mu$s) is in the same range as the Queue Handler figure, which is a good indication that this implementation is internally consistent and similar to other ports of ROME.

The *Perf* program can be left running over a much longer interval. Every 15 minutes it prints out the elapsed time, which can be used as a more accurate measure of the real-time timer programming, for example by running the code over 24 hours. This is also a good test that there are no spurious 'glitches' in the system that manifest themselves only infrequently.

# 14 The SCV64 VMEbus controller

The preceding sections are enough to get you to the stage where you can start writing device drivers and applications for most systems. On some machines, there is still one 'missing link' to accessing devices,

and that is the bus controller. Often, this is a PCIbus controller which gives access to the optional plug-in cards on the motherboard. For the Cyclone boards, it is the SCV64 VMEbus controller, giving access to the external interfaces connected through the VME backplane. This controller is neither a real device driver, nor part of the ROME core, but understanding how to write the ROME code for it is a key to porting ROME to such systems, so I will describe it here.

The main feature that distinguishes a bus controller from other drivers is that there is usually no main process (or queue handler) since it does not itself handle any messages. The module consists of an initialisation routine, and interrupt handler, and a shared library used by device drivers on the bus. It is an 'initonly' process. Also, the initialisation routine must be called before the initialisation routines of any devices connected to the bus, so the module information in RTB must reflect this.

## 14.1 Additions to the Target File

The SCV64 chip has a number of memory-mapped registers in the f000.0000h region. The region has already been set to big-endian and uncached in the *cpu_prologue* routine above, in preparation for this code. The target file contains the usual definitions of where to find the relevent registers, taken from the Cyclone Board User Manual

```
#define VME_SCV64_BASE        0xf0000000
#define VME_IACK0             0
#define VME_IACK1             0xf4000013
#define VME_IACK2             0xf4000007
#define VME_IACK3             0xf4000017
#define VME_IACK4             0xf400000b
#define VME_IACK5             0xf400001b
#define VME_IACK6             0xf400000f
#define VME_IACK7             0xf400001f
#define VME_IPL_REG           0xb4200000
#define VME_XINT0             0
#define VME_XINT1             1
#define VME_VEC_XINT0         0x12
#define VME_VEC_XINT1         0x22
#define VME_SIZE_SHARED SHARED_MEM_SIZE_4M
```

The *IACK* addresses are used handle the different levels of VMEbus interrupt. As is common with bus controllers, multiple device interrupts are combined into a few processor interrupts. In this case, the SCV64 is connected only to external interrupt pins 0 and 1 on the processor. Finally, the amount of shared memory on the system is defined here.

## 14.2 The vme.h header file

The VME bus controller is a combination of interrupt dispatcher, shared-memory manager and DMA controller. The external interface contains routines similar to the *ICU* module, *vme_add_handler, vme_enable_interrupt* and *vme_disable_interrupt*. It also support dynamic plugging of modules, so has a *vme_remove_handler* routine and dynamic interrupt generation through the *vme_generate_interrupt* routine. The header file also exports the interface to the DMA controller 'done' handler, through *vme_add_done* and *vme_remove_done*, and the ability to request memory in the shared space, through *vme_shared_malloc*.

## 14.3 The svc64.c source file

Most of the source code for the controller driver is very specific to the VMEbus architecture, so this description will concentrate on the principles, rather than the details.

### 14.3.1 The init routine

The purpose of the init routine is to prepare the bus so that devices on the remote side (from the CPU) can be accessed and identified. The base address of the VME shared memory can be set by the switches on the front panel of the board. Rather the write a separate target file for each possible setting, the address is calculated dynamically by reading the switch settings;

```
int addr = CPU_IORD1(VME_IPL_REG);
/* setup shared memory */
vme_shared_mem_size = VME_SIZE_SHARED;
vme_shared_mem_base = ((addr >> 3) << 27);
```

Most of the remainder of the routine is writing values into the memory-mapped registers using the various *CPU_IO* macros.

The ROME-specific part at the end of the routine sets up the default VME interrupt handlers and clears the ‘DMA done’ user-specifiable routine:

```
for (i = 0; i < VME_MAX_VECTORS; i++)
{
    vme_proc[i].handler = vme_default_handler;
    vme_proc[i].parameter = i;
}
done_routine_ptr = done_def_routine;
```

It also calls the standard ROME routines to add handlers for the two interrupts connected to the VMEbus, and clears any pending interrupt (caused by the initialisation code or left over from the boot ROM):

```
rome_add_handler(VME_VEC_XINT0, vme_scv64_isr0);
icu_clear_ipend(VME_XINT0);
rome_add_handler(VME_VEC_XINT1, vme_scv64_isr1);
icu_clear_ipend(VME_XINT1);
```

At this point, other devices on the VMEbus should be able to access their register sets and data areas.

### 14.3.2 The interrupt handlers

The SCV64 controller uses 2 interrupts. The higher priority (isr0) is used to critical bus errors and faults while the lower priority (isr1) is used for other faults and device interrupts. Interrupts come a one of seven levels and are accompanied by a ‘vector’ (or device code) written to the VMEbus, which must be read from a level-specific location. For external device interrupts, the handler reads the level from the *VME_IPL_REG* location, the reads the vector from the corresponding *VME_IACKn* address as defined in

the target file. It then uses the vector to index the interrupt table, in the same way as the *ICU* module sets up its array.

One disadvantage of this approach is that there is now a three-level interrupt dispatcher. First through the interrupt handler in the *CPU* module, then to the VMEbus handler and finally to the device handler. However, the alternative is to try to build the VMEbus handler directly into the *CPU* plugin, which make it much harder to separate the processor from the motherboard. Also, since most of the code of the VMEbus handler is directed towards error processing, this is best left outside the main core of the system.

### 14.3.3 The shared library

The implementation of the shared library is straightforward based on the description given in the header file section above.

### 14.3.4 Tracing and Debugging

The VMEbus is not the world's most forgiving bus to program. It is quite easy to lock the bus, or generate bus timeouts, which will cause error-level interrupts to the SCV64 handlers. Debugging these problems usually requires a dump of the register of the controller, and a brief history of the bus activity. The module implements a trace record similar to the standard ROME trace, but for VMEbus events. The reason this was separated from the main trace is partly history and partly so that the VME trace can be viewed separately from the main ROME tracing (which might have progressed beyond the point where the useful VME event was stored).

The trace records are added from the interrupt handlers as they occur, and the display routine, *vme_view_errors*, is written so that it can be called directly from within the debugger (assuming the target was compiled with symbol table support). Although the register settings for the SCV64 chip can be examined from within the debugger (using the **dm.w** command), decoding the various bit patterns to find the error indications is tedious, and prone to error. The module defines another debug-callable routine *scv64_debug* which formats and interprets the registers. This is a very long routine, and it might be possible to conditionally compile it out of a system (using a module option) if the space it takes was needed for other code. On the other hand, VMEbus errors tend to be unpredictable, and having support for this level of information is often valuable. Writing such debug-callable routines is a useful way of keeping the system as modular as possible, since the core debugger knows nothing about the specifics of the VMEbus controller chip.

## 14.4 The SCV64 Module in RTB

There are a few points to note when creating this module in RTB. Adding the description and source files is straighforward. You must also add a process to the process list. The process name does not matter (but "vme" seems like a good choice). After entering the name, the fields for the queue handler and the main process must be cleared to zero, and the initialisation routine set to *vme_scv64_init* to correspond with the actual file.

The other entry that must be modified is the 'link order' field, which defaults to 99 (unset). The value must be smaller than that for any device using the VMEbus. My preference is to use 0–9 for system-critical link positions, 10–19 for bus controllers and 20–29 for early-link devices, which makes '10' a good number for the link order.

At this point, it should be possible to build the module into the system, and try it out.

# 15 Tuning the System

The first rule of tuning is only tune a *working* system.. If it doesn't work reliably, you're not tuning it, you're fixing it, and this is the subject of the next section, not this one. If you're here, I assume that you can run *Perf* for days on end (or overnight at least) with no strange errors, that starting the system works reliably time after time after time and that you don't get any odd 'glitches' (like strange characters on the serial interface) that you have promised yourself you're going to look into 'one day'.

The second rule of tuning is to be *very* suspicious of the changes you make, especially in the low-level routines. Only change one thing at a time, and back it off if it doesn't work (you kept the old version somewhere safe, right?).

The third rule of tuning is don't sacrifice reliability for speed. If you have a 'speedup' that will work 99% of the time (you know, unless the interrupt happens between those two particular instructions, or the timer is just about to roll over) you're not tuning the system, you're *breaking* the system and it's time to go for a walk.

So what can you do to make a system go faster? At this stage, look at the implementation of the interrupt and context handling and the criticality routines. The criticality routines are really simple, with one line of assembler each, and are quite suitable for inlining. The file *icu.h* was modified to inline these two routines:

```
#define rome_start_critical() \
({ int __old; \
   __asm __volatile ("intctl 0, %0" : "=r" (__old)); \
   __old; \
})
#define rome_end_critical(_o) \
   __asm __volatile ("intctl %0, %0" : : "r" (_o))
```

and the whole system was rebuilt. The assembler listing for the protected routine call code in *Perf* was checked to make sure the inlined code was correctly generated. The following extract shows the *intctl* instructions at the start and end of the routine:

```
230 03c8 1E16805C   mov g14,g0
231 03cc 001EF05C   mov 0,g14
232 03d0 001CA865   intctl 0, g5
233 03d4 0030A090   ld _ih,g4
233      F4050000
234 03dc 0108A559   addo 1,g4,g4
235 03e0 0030A092   st g4,_ih
235      F4050000
236 03e8 1514A865   intctl g5, g5
237 03ec 00100484   bx (g0)
```

As expected, this improved the performance figures. That the value for a regular routine call increased is not unusual, since the individual performance figures vary slightly depending on exactly how the instructions are aligned within the cache.

| field | value ($\mu$s) |
|---|---|
| Queue Handling | 6.17 |
| Context Switch | 35.52 |
| Routine Call | 0.36 |
| Protected Routine Call | 0.65 |
| Single Context Switch | 14.68 |

Other options to tune the system involve taking a careful look at the save/restore code for context switching, for example to interleave instructions is possible for maximum parallelism or to place useful code in otherwise wasted branch-delay slots on those architectures that have them. I've already done this once for most of the I960 code, so there isn't anything else much to do here. The one place I did not tune in this example was the first-level interrupt handler. The following code performs the same function as the version listed above (the instructions are identical) but the order is changed to maximise the pipelining:

```
    intdis
    stt g12,(sp)
    movq g0, r4             # Store globals in locals
    lda 12(sp), sp          # Carve out room on stack
    movq g4, r8             # (can only stash 12 registers)
#ifdef CPU_BIG_ENDIAN
    ldob -5(fp), g0         # which intr vector (BE)
#else
    ldob -8(fp), g0         # which intr vector (LE)
#endif
    mov 0, g14              # prevent cx switches
    ld _icu_exception_handlers[g0*4], g1
    movq g8, r12            # this way)
    st g14, _rome_allow_reschedule
    callx (g1)              # Call the handler ; ino in g0
```

This reduced the 'delta' value from 9,020 to 8,920. At this point, I hope you're asking if it was worth the effort. The new code is harder to follow than the old one, and any changes to are likely the break the optimisation. This is probably a good caution against going all-out to tune bits of the core. Given that the timer interrupts occupy under 0.5% of the CPU, this may be a waste of effort. On the other hand, improving the context switching time *was* worthwhile. The next *useful* places to look for optimisations are in the data movement areas, for example by providing fast implementations of *memset* and *memcpy*.

Even though the criticality routines are replaced with macros, the original routines were left in the code. This allows interpreters to use them by looking up the routine name in the symbol table. It's slower than inlining, but interpreting is really slow anyway.

# 16 What if it doesn't work?

So you load up your code, start the system at its entry point and nothing happens, or the 'fault' light goes on on the motherboard, or the ROM reports some catastrophic error, now what?

This is the place to find 'Don't Panic' in large friendly letters. This section is proof that we've been there, and passed beyond it. By now I've probably seen just about everything that can go wrong with a system (but there are always new surprises), including the dreaded puff of orange smoke (which smells really terrible). Where to start really depends on what your working with, and how far you think you're getting with the system.

## 16.1 New Hardware

Probably the hardest system to get working is a brand-new board, especially if it also contains some 'experimental' hardware, and you're also writing the boot ROM yourself. If you've got the money, buy an ICE (In-Circuit Emulator) for your chosen CPU and debug the system with that. If that's beyond your finances, a storage scope or logic analyser will help (to see if there's any bus activity, or if read/write lines are going up and down). However, this manual isn't really about debugging hardware. The easiest way to work with a new board (or an old one) is to make sure your system has some (i.e. one) LED(s) that can be controlled from software, preferably by just writing to a location in memory. It's also useful to put an LED somewhere in the power line, or to check the +5V line with a Voltmeter once in a while — its hard to debug a fried board.

For the most part, I hope you are working with a board that's fairly stable, and with a boot ROM that's been seen to work with other systems, or can talk to its UART and convince you it's alive.

## 16.2 Loader Problems

It's not uncommon that the boot ROM refuses to load the target file you've just made. It's likely that the format generated is not 100% compatible with the ROM loader. If you have a file which *does* work, then you can compare the headers (are they both the same revision level of the selected loader-file format?) and the contents (e.g. does your file contain any 'extended' options such as long addresses in Motorola S-records?). If you have the source for a working file, compile and link it with the ROME toolchain and compare the output to the working version. You might also want to try compiling a tiny assembler file (say to turn on the LED) and loading that. If the file format uses checksums, you might write a little utility on Linux that verifies the checksums, then try sending a file with a bad checksum and see if the ROM detects it.

If the boot ROM loads the file, but fails to run it, then you might want to check the following. Has the file loaded at the right place? If the ROM can display data, does the data at the entry point correspond to the compiled file? Is the file compiled for the right endianness? If the ROM has a disassembler, does the instruction at the entry point look reasonable? Has the target been linked in the right order? You can check with the *target.map* file that *_link_first.o* is at the start of the target. Does the loader report the correct entry point (if it supports target file formats where the entry point can be dynamically varied)? If not, you may need to add a line to the linker file to force the correct entry point.

If you can load a small file (turning on the LED) but not a ROME system, does the loader have some inbuilt limits on file size, or is it timing out over the link (e.g. the serial line) used to download the file? If so, you need to get (or write) a better boot ROM. You might also check that your system does fit into the memory on the board, you may need to reduce the size of data structures or stacks to make it fit.

## 16.3 Initialisation Problems

The most common problems with new systems lie in the assembler code between system startup and calling *rome_start*. This is especially true of systems which have an 'all-or-nothing' mode change in this part of the code (for example the I960 *sysctl* to move the processor tables, or the change to protected mode on the I386). Most often, you will start the system and see — nothing! No output on the serial line, no flashing lights, nothing. There are two or three strategies to cope with this, all of which I've used at one time or another.

1. if you can persuade the serial interface to work from within ROME, you may be able to use the debugger to trace what's going on.

2. if you can control the LED from within the code, you can locate faults by moving the on/off LED position around

3. if you have an external instruction-set-simulator for the CPU you can try running the initialisation code on it and see if it detects the fault.

It might be surprising, but option 3 once solved an 'impossible' problem with context switching where an error ocurred very rarely in a way that could never easily be reproduced, and destroyed most of the system after it happened.

Option 1 is what lies behind that 'alternative entry point' at offset 4 in the startup code above. If you have a system with a working serial interface, then set the *SKIP_INIT* option (so you don't break a working configuration) and try starting the system at the alternate entry point (or re-compile with the first instruction as a no-op). It's actually quite likely that the debugger will run, and you can use it to check memory and disassemble code. If this works, you can move the debug entry point further down the code and check what happens at each stage (for example after clearing BSS to zero).

However, it's more likely that you've broken the serial line too, so the LED is your only option. Even if there isn't an obvious indicator light, sometimes you can be creative to get an indication of what's happening. I debugged the I386 ROME system using the 'motor' light on the floppy drive, which the standard ROM boot code left on after booting the system from a floppy disk, and I could switch off using an *outbyte* instruction. On the Cyclone board, the ROM set the 'run' light on, but when the machine faulted the 'fault' light came on, so at least I could see something was happening.

By moving the LED code around, you can narrow down the range of failing code, or at least detect the point after which everything is bad. That doesn't necessarily mean that it is the preceding instruction that causes the problem, for example clearing the wrong area of memory instead of the BSS may not affect the system until much later, depending on just what was cleared by mistake. The hardest problem is usually the 'all-or-nothing' instruction, in the example above the *sysctl* reset. If any of the tables are incorrect, you won't get very far beyond that code.

The first version of the CVME965 system I wrote used the wrong value for the *REGION* descriptor for the main RAM (because I looked at a manual for an older board by mistake). The result was that the code could not proceed once the new tables were loaded. I solved it by examining the memory-mapped locations for the system registers that contained the versions as set by the ROM, and compared each entry with the value I was about to use in my new tables. It may be that you really *want* to alter some of the values (for example to change processor mode). If you can, you might be able to start by duplicating the ROM values, getting the 'null' mode change to work, and then making the required changes slowly until the system breaks again. Ultimately it comes down to staring at the code and looking for something that shouldn't be there, or is missing.

There is still one little point that might go wrong. One system I had started perfectly and printed the following string:

```
EMORinI laitnisi
                .gn
```

followed by similar 'garbage'. This is clearly not the usual baud-rate problem on the serial line as the characters were all correct ASCII. If you haven't worked it out yet, it's "ROME Initialising\n" in reverse-endianness! The processor fetched instructions in 32-bit wide endian-independent loads, so the code worked fine, but all the character pointer accesses were back-to-front. Changing the endianness fixed that one.

## 16.4 Serial-Line Problems

If entering the debugger directly doesn't work, or the LED shows you returning from the first *rome_kprintf* in the startup code, but there's no output, or the system appears to have translated itself into Icelandic, then you've got problems with the polled-mode serial line.

If disabling the *serial_initp* routine doesn't help, then it's likely one of two things has happened: either the boot ROM reset the serial interface before starting your code (yes, I've seen one ROM that did that), or the code to access the serial interface is corrupted, perhaps because of stores into the wrong locations during initialisation, or is just plain wrong, for example by addressing the wrong memory areas, or the wrong register spacing. A really simple (i.e. 'dumb') example of this happened with my I386 system. The serial interface was configured to use the standard serial port I/O address at 2f8h. However, the BIOS had COM1 set to the 'alternate' address of 3f8h and COM2 (at 2f8h) was disabled. Since I had the serial cable plugged in to COM1 on the back of the motherboard, I didn't get any output. It also helps to try putting a NULL modem in the line in case the system and the display have different ideas about which is the DTE end.

If the serial line works in the boot ROM, and with no *serial_initp*, but fails after re-initialising, at least you know it's the init code. The same is true of the 'Icelandic' output (because of the Thorns and Eths that come in the garbage), which means the baud rate is wrong. Some machines use UART crystals that are not at the usual 1.8MHz. If you can get the interface to work without the init code, on some UARTs you can read out the values programmed into the baud-rate, parity and flow control flags, and see how they match up with the values you are about to put in.

If changing the link order makes the problem go away (try editing the order of the object files in the *ld.input* file) then you have got data or code corruption. Obvious places to look are the data clearing portions of the initialisation code, particularly if you clear memory 'backwards' (starting at the end and working downwards). In one case I remember, I calculated the number of bytes to clear using the *bss* symbols, and then cleared memory using word operations, so clearing four times as much space as I expected. This wiped out a chunk of the initialised data in the system, including, of course, the place where the base address of the serial interface was stored.

If the boot ROM doesn't use the serial port, and you can't get the port work at all, try putting a breakout box on the line and looking for the lights changing to indicate data. If all else fails, treat it as a 'hardware' problem and attack it with a 'scope.

## 16.5 Context Switching

The next most likely place to fail, once you've got through init, is the first context switch in *cpu_scheduler*. There are two separate reasons why this is going to fail. The first is the context switch itself, which will transfer your program into hyperspace if it goes at all wrong. The second is that this is when interrupts are enabled for the first time, and you will handle any pending interrupts that might be sitting around. These problems will manifest themselves in one of two ways. If you're lucky, you will enter the debugger with

a fault message at some strange location. If you are unlucky, when you try to enter the debugger with the '!' key to test the idle process, nothing will happen.

The good news is that you should be able to use some of the facilities built in to ROME to help from this point on. As a first step, you can turn on the *ROME_TRACE_CXSWITCH* and *CPU_KPRINTF_TRACES* options and rerun the system. This should tell you if you are getting past the first context switch, since it should print two lines similar to the following:

```
@0x00000000 proc 0xa005c0c0 code 83 arg a03fec80
@0x00000000 proc 0xa03fec80 code 83 arg a03fda70
```

The first line is the trace entry from *rome_start* just before it calls *cpu_scheduler*. The second line comes from within *rome_await_message* called by the serial process to switch to the idle process. You probably get the first line, but not the second.

Here are some strategies for tackling this. First, you can edit a call to *rome_debug* into the ROME module just before the call to *cpu_scheduler* and use it to check the data areas for the processes (you should be able to use the 'lp' command in the debugger at this point). Second, you can bypass the code in the context switch to enable interrupts, so the serial process, and then idle are entered with interrupts still disabled; this will catch the problem of the bad pending interrupt. Third, you can replace the *ret* instruction (or equivalent) at the end of the context switching code with a call to the debugger. The system should appear to be in the serial process at this point, with a stack frame and register set appropriate to that process. Fourth, you can insert a call to *rome_kprintf* at the start of the serial code to check that it is being entered, and one at the start of *rome_await_message*.

If you get the entry into the fault handler instead, you should try to match up the faulting address with one of the values near the process control block. For example, has the system tried to branch to the stack pointer instead of the entry point? Does the process stack look plausible for initial entry to the process? If the fault address is 'close' to the process entry point, try disassembling from the entrypoint onwards and look for corrupted code. If you find some, then work backwards, by putting calls to *rome_debug* into the code, until you trace the point at which it is corrupted. If the fault appears to be in valid code, then look more closely around that area. One problem I had with another I960 system was a fault just inside *rome_await_message*. All the code looked valid, but the instruction pointer was consistently set just after a *callx* instruction. The call was to (a broken version of) *rome_start_critical*, which was generating a program fault, delayed until after the following *ret*.

Faults just after a call to *rome_end_critical* usually indicate a problem in disabled code or in an interrupt handler. The *TRACE_INTERRUPTS* option may help show which interrupt handler is being called just before the fault. In one case I was getting VMEbus system errors showing up at a *rome_end_critical*. Tracing the interrupt showed the most recent handler was always a VMEbus ethernet card, which had a very long interrupt handler. I added a *VMESTATUS* trace record to the system and recorded the state of the bus on entry and exit to the handler. Indeed it was generating a bus fault. By adding further trace calls within the handler it was possible to narrow the problem down to a incorrectly-initialised pointer. It is, of course, not necessary to run such tests with the *KPRINTF_TRACES* option set, indeed that alone may mask some problems. The usual approach is to use the debugger **trace** command once the fault has been detected to review the events leading up to the crash.

## 16.6 Interrupt Handling

If context switching works with interrupts disabled, but not otherwise, or the '!' character does not enter the debugger, there is a problem with the interrupt code. This is almost as bad as debugging the initialisa-

tion code, but not quite. The first step is to find out if the interrupt handler is getting called at all, either by putting code at the start to turn on an LED or by inserting a call to *rome_debug*. The chances are there is an alignment or offset problem in the interrupt table, so external interrupts are not being vectored correctly. Basically, if enabling interrupts causes the system to fail, and first-level interrupt handler is not being entered, then there is a problem with the interrupt table configuration. The only solution is to stare at the code until enlightenment dawns, aided perhaps by looking at the actual system with the debugger to spot discrepancies between what you thought you wrote and what is in the machine.

If you've made it into idle, but '!' does nothing, then the problem is either the same as above, or the UART interrupt is not correctly configured. Unless you're *really* desperate, don't put a *rome_kprintf* or equivalent into the serial interrupt handler, it's a good way to lock up the system with infinite interrupts (because I tried it). You might want to put some output (or a call to *rome_debug*) after the serial initialisation code, to check that the handler is installed in the correct vector and the interrupt mask value is set to allow the interrupt through. You might also check in the Build (or in *Hardware.h*) that the *ENTER_DEBUG* option has the right value of '!'. If you still have the LED code in the interrupt handler, you should see the light go on when you press any key; if not, the character-received interrupt is not enable on the UART. You can try to enable *ROME_TRACE_INTS* and remake the serial module (and set *CPU_KPRINTF_TRACES* if you're brave) and see if it really makes it into the handler. If it doesn't, then the interrupt vector is being decoded incorrectly. If it does, then the input character is being read or tested incorrectly.

Once you're passed this point, the only other problem is likely to come in *Perf* after the first round of testing and the "Queue Handler" message is printed. This will be the first time that a process context switch will occur as a result of an interrupt — when the final character of the string is printed, the buffer is returned to the *Perf* process which is waiting for it, during which time the idle process has been scheduled. This will test if the *IRSched* code, or its equivalent, correctly saved the context information such that it can be restored. Hitting a problem in this code was one of the more surprising bugs I found while porting the ROME core, since I really wasn't expecting it this 'late' in the program. It just goes to show, the code doesn't work until until it's actually been run

# 17 And Finally...

Everything works, the system is tuned and the performance figures look good. All you're going to do now is document the new modules and submit them to the server, right? If, like me, you'll admit to the mistakes as well, you might want to add some more hints to the previous chapter, about how *you* got your ROME system working.