The ROME Programmers' Guide

May 1, 2001

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

mailto:rome-admin@lists.sourceforge.net

You can get the current version of this document at http://rome.sourceforge.net

Contents

List of Figures

1 Introduction

This document presents a structured walk-through of the use of the ROME Operating System. It gives examples of use of the external support tools including the ROME Target Builder (*RTB*). It describes the API to the kernel and to the dataflow abstractions implemented in the Run-Time library. The document also uses a number of 'hands-on' examples of modules in a tutorial manner, gradually to integrate the various standard components of ROME.

1.1 Summary of Features

A complete ROME system is built up from a number of modules according to the required application. Some modules are specific to a particular piece of hardware (the cpu plug-in, and device drivers), some are machine independent and usable across a wide range of applications (for example the ROME core and the Console multiplexer) and some are application specific.

Internally, ROME uses a message-passing system for inter-process communication. ROME defines a standard set of messages, but modules are able define their own messages for specific functions. Most of the details of individual messages are hidden from high-level programs by interface libraries (including an implementation of the standard 'C' runtimes). However lower-level modules (including device drivers) must understand the format of messages in order to process them.

The ROME system is linked into a single naming (and addressing) space. This means that most interfaces are available through shared libraries, but places restrictions on the variable names which may be used.

To manage this system the ROME toolkit contains a configuration utility *rtb*. Compilation is performed using *gcc* as a cross-compiler, with the appropriate assembler and linker.

When described in 'Reference Manual' format this combination of tools, environment, libraries and interfaces may appear daunting, though the underlying principles are deliberately simple. This document seeks to overcome that obstacle by presenting a walk-though of ROME features in a sequence of worked examples, which can be built, loaded, and executed on readily-available hardware. This guide should be used in conjuction with the other ROME manuals and the reference pages for all the modules currently available in ROME.

1.2 The minimal ROME system

The smallest ROME system that it is possible to construct contains five components:

1. The 'system' shared library, implementing the eight basic core routines for the message-passing and memory management functions, and the "idle" process, run by the scheduler when no other process is runnable;
2. The CPU-dependent plug-in, containing the context switching code for inter-process communication;
3. The interrupt dispatcher;
4. The system-level debugger (optional, but highly recommended);
5. The 'C' runtime library and core interface library for the Standard MessageSet.

These five elements are shown as part of the ROME base system in the process description figures.

1.3 Layout Conventions

In the tutorial sections, examples of user input are used. Such input is distinguished by a **`bold`** `typewriter` font. System output is in regular weight. 'C' examples are formatted in regular and **bold** weights.

The examples of system builds, and directory listings are copies 'asis' from test systems used to verify the build. Naturally, the output produced at different times on different machines will cause this output to vary.

1.4 Setting up the ROME Environment

Before you can run the programs in the tutorial, you have to install the ROME system on a local machine. The ROME environment is divided into an optional CVS source tree, local 'project' directories and toolkit binaries. At this stage is it assumed that you have already downloaded and installed the ROME Target Builder (*RTB*) according to the RTB User Manual, and have its binary on your search path. The project directories can be located anywhere in a (writable) filing system

You may need to consult the "Getting Started" guide, or your local adminstrator to find out how the ROME sources are managed in your environment. The sources will either be in a local copy of the tree and be accessible through the RTB *import* command or be in a CVS repository and be accessible through the RTB CVS dialogues. Before you start, you will need to configure your personal RTB preferences.

All these examples are designed to run on an Intel-based PC, with at least a 486-class CPU. It is also assumed that the development platform is PC-based. The tutorials boot directly from a floppy disk on the target machine (which means that the BIOS on the target must have 'boot-from-floppy' as the primary boot option). All the output is directed to a 9600-baud serial line at 0x2f8 (usually 'COM1'). The support toolkit contains a program, *xtip*, that can be used to display this output and send input keystrokes to the system. The reasons the examples use the serial interface (rather than the PC keyboard and screen) are to simplify the programs, and better to represent 'real' embedded applications, which are usually debugged through a UART interface, rather than an attached screen and keyboard.

1.5 Project directories

ROME systems are built within a host filing system, in a personal or project directory. RTB uses this directory as the place in which to create a new system, and is designed to work with a completely empty initial directory. However, the configuration information for each project must be stored somewhere. These tutorials assume there is a single directory being used for all these projects. The top-level of this directory holds project information and the individual projects are created in sub-directories. At this point you should set the *projects* field in the RTB *Preferences* dialog to this directory (e.g. */home/leslie/rome_tutorial*). This completes the environment setup, as RTB automatically creates all directories that are needed. Once you have done this, you can move into the projects directory:

```
cd rome_tutorial
```

2 Tutorial I: Hello World

This first tutorial uses pre-defined sources to introduce the configuration and compilation mechanism. It builds a simple loadable system which prints *hello world* on the console output. The process structure for

this system is shown in figure 1. Apart from the minimal core described in chapter 1, this system includes a driver for the system's UART (serial line). Two non-driver processes complete the system, the console process and the hello process.

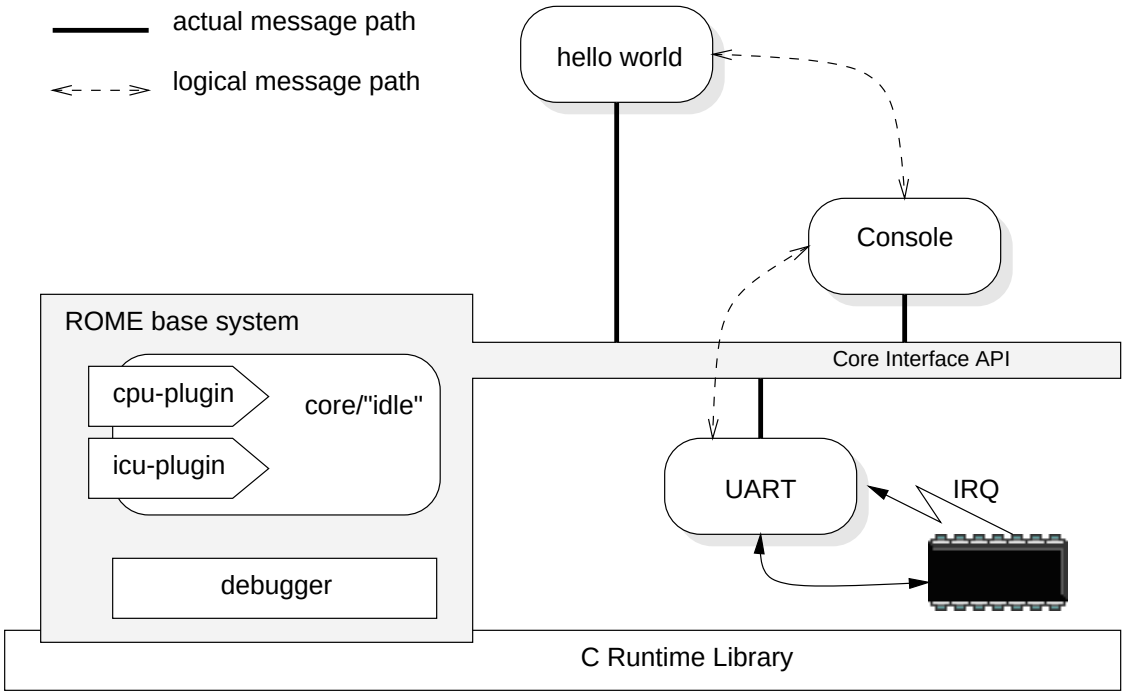

Figure 1: Module Structure for Tutorial I

2.1 Step 1. Build Input

The first stage is to create a project description using *RTB* in the directory created in chapter 1.

In *RTB*, create a new project description for the *hello world* program and check out (or import) the following modules, according to the CVS hierarchical structure (if you are importing the modules, you do not have to provide the dotted class ID prefix):

Target: SPB450

MessageSets: Standard, Console

Modules: Kernel.PlugIn.CPU_I386, Driver.Interrupt.ICU_I386, Kernel.ROME, Library.ROME_IF, Driver.Serial.SER
Application.System.CONSOLE, Library.CLIB, Application.Demo.HELLO

This describes the basic hardware configuration and brings in the core scheduler and drivers needed for the tutorial. When examining the individual processes, note that the *hello* Process line has the 'use stdio' flag set, in contrast to the other processes.

2.2 Step 2. Generating the Source Tree

Click on the 'Generate Build File' button to run the system builder program and then check the dialog popup for any errors. The result should be a top-level directory listing looking something like this:

```
cd hello_world
ls -l
total 7
drwxr-xr-x  4 leslie leslie   1024 Apr 2 17:30 MessageSets
drwxr-xr-x 13 leslie leslie   1024 Apr 2 17:35 Modules
-rw-rw-r--  1 leslie leslie   3435 Apr 2 17:34 Project.rtb
drwxr-xr-x  3 leslie leslie   1024 Apr 2 17:30 Targets
```

Each of the components types has a sub-directory below the project directory, which also contains the project description file.

```
cd Modules
ls -l
total 14
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 CPU_I386
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 Clib
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 Console
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 Hello
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 ICU_I386
-rw-rw-r--  1 leslie leslie   1849 Apr 2 17:35 Makefile
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 ROME
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 SERIAL_UART16550
drwxr-xr-x  2 leslie leslie   1024 Apr 2 17:35 Symbols
drwxr-xr-x  2 leslie leslie   1024 Apr 2 17:35 include
drwxr-xr-x  2 leslie leslie   1024 Apr 2 17:35 init
-rw-rw-r--  1 leslie leslie    873 Apr 2 17:35 ld.input
drwxrwxr-x  3 leslie leslie   1024 Apr 2 17:35 rome_if
```

In the *Modules* sub-directory, each source module has a directory created for it, and there is an configuration-specific *include* directory for the header files appropriate to this build. The other directories, *init* and *Symbols*, are created to hold the process-specific initialisation files and the debugger symbol table, respectively.

Finally, *Makefile* and *ld.input* are input files to the following stages.

2.3 The Program Source

Before proceeding to compile the system, this section shows the source file used as the application to print *hello world*. The file is *hello.c* in the *Hello* sub-directory.

```
ls -l Hello
total 5
drwxrwxr-x  2 leslie leslie   1024 Apr 2 17:34 CVS
-r--r--r--  1 leslie leslie   1001 Aug 26 2000 Hello.rtb
-rw-rw-r--  1 leslie leslie    851 Apr 2 17:35 Makefile
-r--r--r--  1 leslie leslie   1759 Jul 18 2000 hello.c
```

As this module was retrived from the local CVS repository, the source file is a read-only copy of the main CVS file. It contains the following source code:

```
#include <stdio.h>

void hello_process(void)
{
    printf("hello world\n");
}
```

This is essentially a standard 'C' program using the run-time library *printf* routine. One difference from most UNIX implementations of the same program is that ROME does not use the *argc* and *argv* parameters to processes. Also, since ROME uses a single shared namespace, routine names like *main* are rarely appropriate. Most externally-visible names are composed of a module name (*hello*, here) and a module-specific local name. The main exceptions are standard routines (like *printf* in the runtimes) and variables declared to be **static**.

2.4 Step 3. Generating the target file

The directory is set up ready to run the *make* utility (which is not part of the ROME toolkit, but should be on a utilities path for the shell). Note that the exact output may vary between builds:

```
make
cd CPU_I386; make
. . .
cd ROME; make
. . .
. . .
cd Symbols; make
. . .
ld -T ld.input -Map=target.map
```

make runs the C compiler and assembler in each of the module subdirectories, then runs *ld* to produce the top-level target. This adds two files to the main directory:

```
-rwxrwxr-x  1 leslie leslie 87363 Apr 2 17:59 target
-rw-rw-r--  1 leslie leslie 19172 Apr 2 17:59 target.map
```

target is the downloadable image, and *target.map* is the linkage address map and symbol table for that image.

2.5 Step 4. Installing the Image

In order to download the image, it must be copied onto a bootable floppy disk, with a boot loader at the start of the file. This is done using the *install* rule in the Makefile:

```
make install
. . .
getprog target target.b
  sh[1]: name .text type PROGBITS flags 6 addr 10000 . . .
  sh[2]: name .rodata type PROGBITS flags 2 addr 198a0 . . .
Pad 15 bytes
  sh[3]: name .data type PROGBITS flags 3 addr 1da00 . . .
(dd if=/p/sa/rome/v1.0/Tools/bin-8086/boot1.b bs=512 conv=sync;
                dd if=target.b) > boot.b
0+1 records in
1+0 records out
138+1 records in
138+1 records out
dd if=boot.b of=/dev/fd0 bs=512 conv=sync
139+1 records in
140+0 records out
```

The *getprog* program (supplied as part of the ROME toolkit) extracts the binary parts of the image from the target file. The first *dd* command pads this out to an integral number of sectors. The second *dd* command adds the boot loader to the front and writes the whole image out directly to a floppy disk. The *boot.b* loader is also supplied in the toolkit.

2.6 Step 5. Loading and Executing the Image

The floppy disk should be placed in the target machine and the machine reset (or powered on). The following output should appear on the *xtip* console.

```
ROME Initialising.
Copyright 1997 NEC USA Inc.
Built by leslie on pc-rome.ccrl.nj.nec.com at Mon Apr 2 18:21:14 2001
Starting the Scheduler.....
hello: hello world

hello: terminated
```

The lines are printed by the downloaded image. The ‘Built’ line gives lists the username, hostname, date and time at which the *build* was invoked from RTB to generate the configuration for this image. This information helps ensure that the correct image is being loaded.

The ‘hello world’ string is preceded by the name of the process which generated it. This is one of the functions of the ‘Console’ module included in the image.

3 Tutorial II: Echo

The previous tutorial used only pre-defined source modules to produce an image. In this tutorial a new application-level module is created, showing how the module is integrated into the build environment. This module, *Echo*, will just copy lines of input on *stdin* to *stdout*. However, the main purpose of this module is to describe integrating a new component into the ROME build environment. The process structure is shown in figure 2. The only difference between this figure and the previous tutorial is that the “hello” process is replaced by the “echo” process.

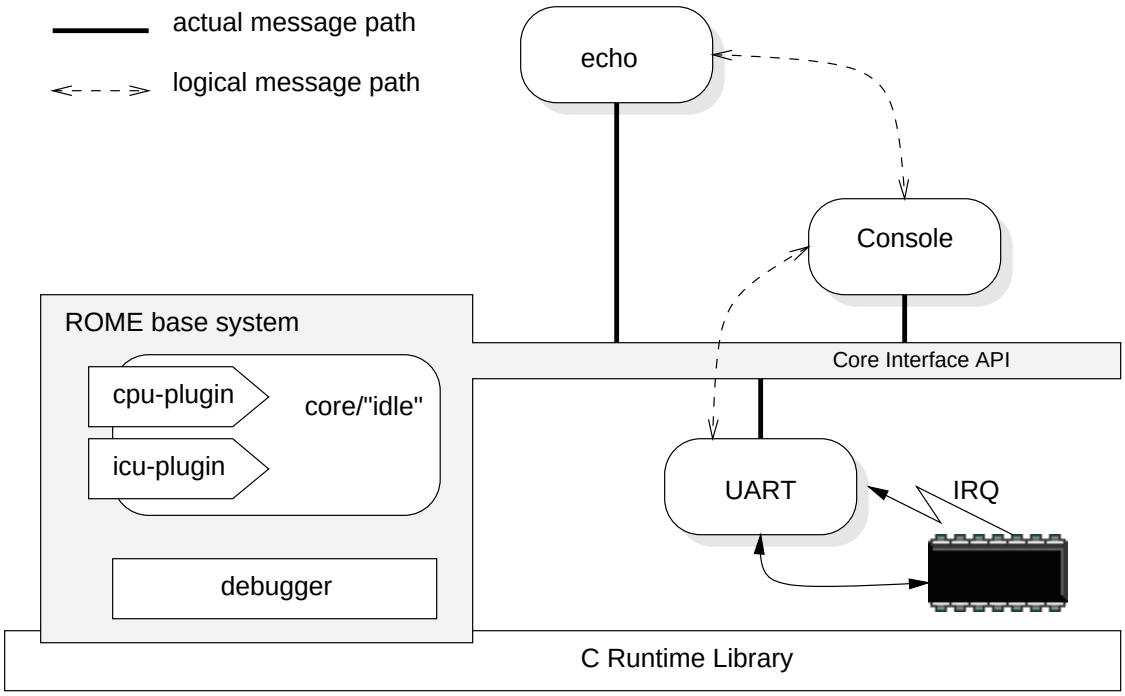

Figure 2: Module Structure for Tutorial II

3.1 The module directory

The names following the *Get* or *Import* commands in the RTB dialogues are the names of modules, for example *CPU_I386* and *Hello*. In ROME, a module is just a directory within the filing system. Modules must be copied into the project directory before they can be used in the project.

For this tutorial, create a new directory under the *rome_tutorial* directory called *Sources* and create a sub-directory within it called *Echo*. This will be the development directory for the new module.

```
mkdir rome_tutorial/Sources
mkdir rome_tutorial/Sources/Echo
cd rome_tutorial/Sources/Echo
```

This structure, with individual module directories under a top-level directory, follows the pattern of the ROME CVS tree arrangement.

3.2 The source file

The executable code of this example is quite straightforward. Using the standard C runtime routines a buffer is alternately filled and emptied. The application runs as a single ROME process from the routine *echo_process*. The complete source for the *echo.c* file is then as follows:

```
#include <stdio.h>

void echo_process(void)
{
```

```
        char buffer[256];
        while (fgets(buffer, 255, stdin))
        {
            fputs(buffer, stdout);
        }
    }
```

Create this file using your favourite edit, and this completes the Echo module source (easy, isn't it).

3.3 Setting up the Project

This step is almost the same as in the previous tutorial. Start *rtb* from the *Sources/Echo* directory and create a new project *Echo*. Check out (or import) the same set of of components as in tutorial I, except for the *Hello* module (if you are importing the modules, you do not have to provide the dotted class ID prefix):

Target: SPB450

MessageSets: Standard, Console

Modules: Kernel.PlugIn.CPU_I386, Driver.Interrupt.ICU_I386, Kernel.ROME, Library.ROME_IF, Driver.Serial.SERIAL_U
Application.System.CONSOLE, Library.CLIB

3.4 Creating the Module

Staying within the *Echo* project in RTB, click on the *New Module* option, then edit the *NewModule0* module in the Modules list as follows:

1. Change the module name to Echo
2. Set the Class ID to Application.Demo
3. Under 'Source Files' add a C file and select echo.c (assuming you started RTB in the Sources/Echo directory). Select the 'Copy to module' option. You could also create the Echo module and the echo.c file directly in *Module/Echo* in which case you would use the 'Just add file' option.
4. Under 'Processes' add a new process named 'echo'. Remove the entries for the Init Function and Queue Handler, leaving only the Main Function named 'echo_process'. Set the priority to 1 and the stack size to 4k and (most important) check the 'Process uses stdio' box.

This completes the creation of the new module for the project. The status of the module should be 'local only' in the project listing. Check the box to the right of the CVS status to include the module in the build. At this point, the *Modules/Echo* directory in the *rome_tutorial/Echo* tree contains two files:

```
total 2
-rw-rw-r-- 1 leslie leslie 939 Apr 3 12:19 Echo.rtb
-rw-rw-r-- 1 leslie leslie 155 Apr 3 12:15 echo.c
```

The source file was copied from the development directory, and the *Echo.rtb* file was created by *RTB* to hold the project description. In this case, the source file is writable, because it was not created through a CVS checkout.

3.5 Build

As before, click on the *Generate Build File* button in RTB, producing the local module directories and the top-level command files. Mostly these are the same as the previous example, but there is now an Echo directory and no Hello directory.

We can trace through some of the build procedure for this module using the files in these directories. Towards the end of the *Output Summary* popup is the record of the processing done for the *Echo* module:

```
Creating MAKEFILE '/home/leslie/rome_tutorial/echo/Modules/Echo/Makefile'
```

The *build* operation has also generated an entry in the top-level *Makefile* for the Echo module:

```
mod_Echo: Echo/echo.c
    cd Echo; make
```

The dependency list is derived from the list of files in the *Echo.rtb* file.

Within the *Echo* directory, the *Makefile* contains a dependency on the *echo.o* file, derived from the 'C' source file name added to the module description. The rest of the *Makefile* is a set of rules used to control the compilation. These depend on the target CPU for the project, and come from the *Target/SPB450* file. Setting up these rules is described in the ROME Porting Guide

The procedure is completed by the directive in the *ld.input* file to include the object in the final target image:

```
INPUT ( Echo/echo.o )
```

All that remains to complete the chain is to start the process within the executable image. The *Process* information in the RTB dialogue has become a initializer in a data structure which is compiled into the image. This is the *init.c* file in the *init* module, which is created automatically during build:

```
extern void echo_process(void);
. . .
    {"echo", 0, echo_process, 0, 1, 4096, 1, 0},
```

Although there are other options and directives which may be used, all modules follow this same pattern from a source tree into a local directory with entries in the Makefiles and the process initialisation table.

3.6 Make, Load, Run

The target is made using *make* as before, and then copied onto a floppy using *make install*, and booted into the test machine:

```
ROME Initialising.
Copyright 1997 NEC USA Inc.
Built by leslie on pc-rome.ccrl.nj.nec.com at Tue Apr 3 12:45:27 2001
Starting the Scheduler.....
```

At this point the system is waiting for input. All keyboard input goes through the "console" process. In the previous example, console output was shown, with the process name being pre-pended to output lines. The same scheme works for input, the first token on the line is used as a process name.

```
echo foo
echo: foo
echo this is a test of the echo system
echo: this is a test of the echo system
```

and as before, output appears with the process name before the line.

The process is in a continuous loop waiting for input, it never terminates and so the 'terminated' message seen in the previous tutorial does not appear.

4 Tutorial III: Perf

This tutorial introduces the ROME messaging system using the low-level performance measuring tool, *Perf*. Although *Perf* is an important tool for categorising system performance (see the use of *Perf* in the ROME Porting Guide), the main focus of the description here is on how it handles inter-process communication.

The real key to the ROME system lies in efficient inter-process communication. The system scheduler is built around a request–response messaging system. A process wishing to use the services of another process in the system locates the destination process using the core routine *rome_find_queue*. The sending process generates a request in the form of a standard message block and queues it for the destination process using the core routine *rome_send_message*. The destination process receives the message into its processing loop by calling *rome_await_message*. Once it has handled the request, the message is returned to the originator using *rome_reply*, a core macro which calls *rome_send_message* to enqueue the reply to the originator. The originator receives the reply with *rome_await_message* and the cycle is complete.

In some cases, requests may require very little immediate processing by the receiver. Examples include adding a 'read' request to a queue of requests awaiting the arrival of data, or 'status' messages that can be answered immediately. For these operations the overhead of the pair of context switches far exceeds the processing needed to handle the request. ROME provides a 'fast path' interface for a receiving process to handle such messages in the context of the sending process. This routine, termed a 'Queue Handler', is called as a subroutine from the core before the message is added to the destination process' queue. The handler must report one of three actions back to the core: (a) the message has been completely processed by the handler and no further action is required; (b) the message has been processed by the handler and the destination field updated, the core should pass the message to the new process; or (c) the message should be queued for the main process.

One frequent operation for a queue handler to perform is to add a message to a queue data structure shared with the main process (or an interrupt handler). In these cases, the queue handler often requires an interlock on the queue, to prevent simultaneous updates which may destroy its integrity. The ROME core implements a single form of interlock, the critical section, which is a block of code guaranteed to run to completion without pre-emption by another process or handler. Since the *rome_send_message* routine, which calls the handler, runs as a critical section to protect the system scheduler queues, the Queue Handler routine automatically runs interlocked.

As an alternative to messaging, for operations that can complete immediately, a simple shared-library interface may suffice. This incurs only the overhead of a procedure call and return. In cases where the library must access shared data, the routine must run in a critical section. This adds the overhead of one call each to *rome_start_critical* and *rome_end_critical*.

Depending on the architecture of the machine a particular function may be better implemented with one solution from this range rather than another. For example, the Queue Handler overhead is added to all messages sent to the process. If the handler only processes very few messages out of the total number sent to the process, it may be less expensive, in the long term, to omit the handler function and allow those few messages to pass to the main routine.

The *Perf* process is used to calibrate a system for these costs. It is used here to introduce the messaging concepts described above and as an important introduction to the timing and performance issues that will govern some of the design of a full system.

4.1 Perf Functionality

Perf measures the amount of real time taken for various low-level operations within the core of the ROME system. It uses the system timer module to request a message every 10 seconds. In each 10-second interval it performs one of a set of four tests by counting the number of times a message can be passed through the system before the 10 seconds elapses. A simple division then given the elapsed time for a single operation. The four operations are:

(a) Sending a message which is returned immediately by the Queue Handler, and receiving the reply.

(b) Sending a message to the queue of the destination process, which generates the reply, and receiving the reply (2 context switches)

(c) Operation (b) plus a single routine call

(d) Operation (b) plus a routine call which runs a critical section.

Subtraction of (c) from (b) gives the routine call overhead, subtraction of (d) from (b) gives the overhead for a protected routine call. Obviously, measurements obtained from the subtraction of two similar numbers are not particularly accurate, however, the results are quite consistent over a wide range of machines and architectures.

There is also a longer-running timer process included in the *Perf* module. By measuring the real time of the system, the interval timer can be calibrated over a number of hours (for example overnight). This gives a measure of the accuracy of the interval timer on which these measurements are based.

The process structure is shown in figure 3.

4.2 The Perf Messageset

Individual messages are identified by in ROME two numbers, one for the message and one for the reply. Message numbers are generated by the RTB tool when messages are defined. There is a ‘standard’ set of messages pre-defined in ROME for managing dataflows; these are the subject of the tutorial V. In

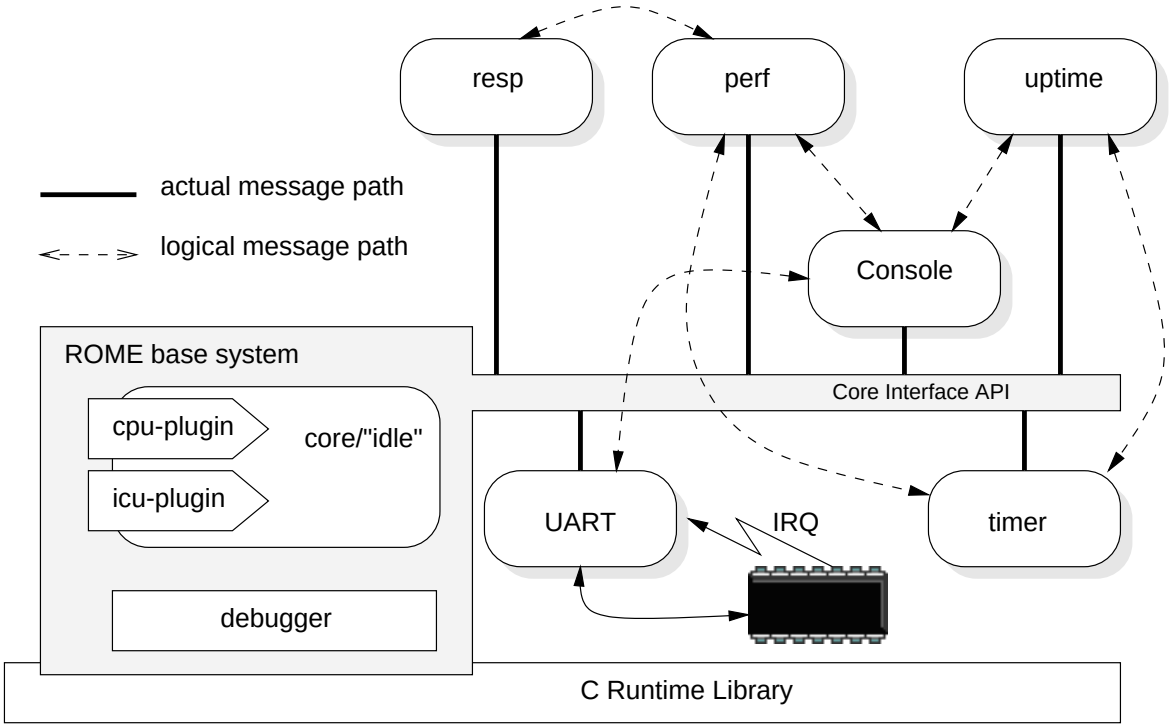

Figure 3: Modules Structure for Tutorial III

this tutorial a pair of private messages are used between two processes, and a message defined by another process is used through a library interface. Because messages are used to communicate between processes, many modules may share the same message definitions. Since messages do not 'belong' to processes in any obvious way, the definitions of messages are grouped into separate components called *Message Sets*. Here, two private messages are defined in the Message Set for Perf. The messages are called *TESTQ* and *TESTCX*, and each takes a single (dummy) argument. This module is available in the master source tree (*Modules/Perf*), and the messageset is in *MessageSets/Perf*.

```
Message TESTQ
{
    int arg1;
}
Message TESTCX
{
    int arg1;
}
```

The *arg1* fields are not used in this example.

When this is processed by *RTB*, the four symbols *ROME_M_TESTQ*, *ROME_M_TESTCX*, *ROME_R_TESTQ* and *ROME_R_TESTCX* are generated to identify the messages and the corresponding replies, and the parameter lists are turned into type definitions.

4.3 Generating the Source Tree

This tutorial uses only existing modules. Create a new project *Perf*, and check out (or import) the following set of of components(if you are importing the modules, you do not have to provide the dotted class ID prefix):

Target: SPB450

MessageSets: Standard, Console, Perf, Timer

Modules: Kernel.PlugIn.CPU_I386, Driver.Interrupt.ICU_I386, Kernel.ROME, Library.ROME_IF, Driver.Serial.SER
Driver.Timer.timer_pc, Application.System.CONSOLE, Library.CLIB, Application.Demo.Perf.

When the project is built, as part of the build, the file *Messages.h* is created in the *include* directory. At the end of this file are the definitions use by Perf for its messaging operations:

```
typedef struct
{
    int arg1;
} ROME_T_TESTQ;
#define ROME_M_TESTQ 0x00020102
#define ROME_R_TESTQ 0x00020103
typedef struct
{
    int arg1;
} ROME_T_TESTCX;
#define ROME_M_TESTCX 0x00020104
#define ROME_R_TESTCX 0x00020105
```

Reply codes are distinguished from the corresponding request by having the *ROME_REPLY_BIT* (0x01) set. Note that requests have even codes, and replies have odd codes.

4.4 The Source Code

The code for Perf will be presented in smaller sections to emphasise the functionality of the individual routines and procedure calls. The system comprises three processes: “perf” acts as originator for the various messages; “resp” is the responder process; and “uptime” measures the length of time it has been running. Of the three, “uptime” is the simplest and will be described first.

4.4.1 The uptime process

This process wakes up once every 60 seconds:

```
void up_process(void)
{
    timeout(up_timer, (caddr_t)0, 60*HZ);
```

The documentation for the Timer module describes the shared library interface for requesting interval timeouts. The *timeout* routine sends a message to the timer module. The reply is delayed for the amount of time specified on the call.

```
        while (1==1)
        {
            ROME_MESSAGE *mp = rome_await_message(0, 0);
```

So, having sent an initial timeout request, the process just sleeps inside *rome_await_message* waiting for the reply. The reply comes as the return value from *rome_await_message*.

```
            switch (mp->opcode)
            {
                case ROME_R_TIMEOUT :
```

The *opcode* field in the message identifies the message type. The **switch** statement handles the (only) message type received by this process (*ROME_R_TIMEOUT*)

```
                    timer_tmhandler(mp);
                    timeout(up_timer, (caddr_t)0, 60*HZ);
                    break;
```

Usually the timeout mechanism is used to call a registered routine, in this case *up_timer*, when the timer expires. This is implemented by the *timer_tmhandler* routine. In a production system, the call to *timer_tmhandler* is usually hidden behind a generic message handler mechanism (explained in tutorial V), but here it is made explicit. The main process calls the *timer_tmhandler* routine as required, before re-issuing the timeout request.

```
                }
            }
        }
```

Finally, all this is enclosed in a ‘forever’ loop, so the process runs as long as the system is up.

4.4.2 The uptime timeout routine

The routine counts the number of minutes:

```
void up_timer(caddr_t dummy)
{
    mins++;
    if  (mins == 60)
    {
        hours++;
        mins = 0;
    }
    if  (0 == mins % 15)
    {
        printf("%d:%02d\n", hours, mins);
    }
}
```

and prints a message every 15 minutes.

4.4.3 Data for the uptime process

The uptime part of the process is completed by the two data definitions:

```
static int hours = 0;
static int mins = 0;
```

and the inclusion of the header files for the library and message interfaces:

```
#include <Messages.h>
#include <stdio.h>
#include <timerlib.h>
```

The contents of the *Messages.h* file produced by *RTB* is described above.

4.4.4 The Perf main process

This routine shows how to construct a low-level ROME message. The routine uses a variable, *state*, to identify which measurement is being made, and uses this variable to select the appropriate message code:

```
void perf_process(void)
{
     ROME_MESSAGE m;
     ROME_PROCESS *id = rome_find_queue("resp");

     timeout(perf_timer, (caddr_t)0, 10*HZ);
```

The logic for establishing and handling the timeout is the same as for the uptime process, the only difference being that a timer is requested every 10 seconds.

```
     m.dest = id;
     m.src = rome_this_ptr;
     m.opcode = (state == 0 ? ROME_M_TESTQ : ROME_M_TESTCX);
     m.priority = rome_this_ptr->prio;
     rome_send_message(&m);
```

The main focus of this code is low-level messaging. The process uses the *rome_find_queue* routine to locate the responder process, and saves the return value in the *id* variable. In order to prepare a message, the core requires that three fields be set in the message, these are:

(a) the *dest* field, which identifies the destination process for this message (in this case the *id* value).

(b) the *src* field, which identifies the originator of the message. Here, the process uses the core global variable *rome_this_ptr*, which always contains the identifier of the currently-running process in the system.

(c) the *priority* field, which specifies the scheduling priority of this message, relative to other messages in the system. The convention is for a process to generate messages at its current scheduling priority, which is found at the *rome_this_ptr->prio* location.

one further field is required in all messages, though it is not used in the core:

(d) the *opcode* field identifies the function of this message. Note that originated messages have *_M_* codes, the reply has the corresponding *_R_* code.

With these fields set, the message can be sent to the destination process using *rome_send_message*.

```
        while (1==1)
        {
            ROME_MESSAGE *mp = rome_await_message(0, 0);

            switch (mp->opcode)
            {
```

in this process, the reply can be to one of three possible messages: either the timer message; or one of the two *perf* messages.

```
                case ROME_R_TIMEOUT :
                    timer_tmhandler(mp);
                    timeout(perf_timer, (caddr_t)0, 10*HZ);
                    break;
```

timer replies are handled in exactly the same way as in *uptime* but calling a different handler routine.

```
                case ROME_R_TESTQ :
                case ROME_R_TESTCX :
                    mp->dest = id;
                    mp->src = rome_this_ptr;
                    mp->opcode = (state == 0 ? ROME_M_TESTQ : ROME_M_TESTCX);
                    rome_send_message(mp);
                    break;

                default :
                    rome_fatal("Bad message");
            }
        }
    }
```

The Perf messages execute in a continuous loop, when the reply is received the message fields are reset and the message is sent off again, possibly with a different operation code, depending on the *state* variable.

4.4.5 The resp Queue Handler

All messages sent to the *resp* process first go through the queue handler:

```
    int resp_qhandler(ROME_MESSAGE *m)
    {
        if  (m->opcode == ROME_M_TESTQ)
        {
            qh++;
            rome_auto_reply(m);
        }
        return ROME_NOT_HANDLED;
    }
```

The routine filters out the *TESTQ* message, using the *rome_auto_reply* macro to send it back to its destination. This macro inverts the *src* and *dest* fields in the message, changes the *opcode* to the *_R_* form and returns the message back to the core indicating that the core should schedule the message on the new queue.

4.4.6 The resp main process

The responder main process uses the same *state* variable as Perf to determine which test to run. Test 0 is the Queue Handler test (which never appears here at all), test 1 is the basic message-handling measurement, test 2 is the procedure call test, and test 3 is the protected procedure call test.

```
void resp_process(void)
{
    while (1==1)
    {
        ROME_MESSAGE *m = rome_await_message(0, 0);

        ph++;
        if  (state == 2)
        {
            resp_proccall(m);
        }
        else if (state == 3)
        {
            resp_iproccall(m);
        }
        rome_reply(m);
    }
}
```

rome_reply is the equivalent macro to *rome_auto_reply* for use in code other than Queue Handlers.

4.4.7 The resp procedures

The responder part of the code is completed with the two routines to measure procedure call times:

```
void resp_proccall(ROME_MESSAGE *m)
{
    ch++;
}
void resp_iproccall(ROME_MESSAGE *m)
{
    int old = rome_start_critical();

    ih++;
    rome_end_critical(old);
}
```

Although the *m* parameter is not used in either of these routines, the overhead of passing a parameter into a routine gives a better measure of the true cost of a routine. The *resp_iproccall* routine uses the core routines to establish a critical section around the counter update.

4.4.8 The perf timeout routine

The timeout routine which runs as part of the perf process evaluates the performance calculations and updates the state variable from test to test.

```
void perf_timer(caddr_t dummy)
{
    if  (state == 0)
    {
        qh = qh / 10;
        qh_t = 100000000 / qh;
        printf("Queue-Handling %d.%02d\n", (qh_t / 100), (qh_t % 100));
        qh = 0;
        state = 1;
    }
    else if (state == 1)
    . . .
}
```

The appropriate counter variable is divided by 10 to yield the number of operations per second. This is converted into micro-seconds and hundredths of micro-seconds, which are then printed.

4.4.9 Variables

Finally, the executable part of the source file contains definitions for the variables shared between the perf and resp processes.

```
static int state = 0;
static int qh = 0;
static int ph = 0;
static int ch = 0;
static int ih = 0;
static int qh_t = 0;
static int ph_t = 0;
static int ch_t = 0;
static int ih_t = 0;
```

Since the counter variables are shared between two processes a natural question may be why two of them (*ph* and *ch*) are updated outside critical sections. The usual requirement is indeed that an interlock would be required. However, because the two processes exchange messages at the same priority, neither is pre-emptible by the other, so each process runs to completion before the other is scheduled and no interlock is required.

4.4.10 The RTB entry

The only unusual aspect of this module is that the RTB entry contains three process lines, one for each process in the file. The resp process registers its Queue Handler as the third of the standard process entry points.

This emphasises the point that there may be more than one process associated with a module, even within a single source file. Each process requires the corresponding entry in the RTB module control file.

4.5 Running the image

The image can be compiled and installed as usual. When it is booted and started, the image enters a loop printing the four lines of performance metrics, forever:

```
ROME Initialising.
Copyright 1997 NEC USA Inc.
Built by leslie on pc-rome.ccrl.nj.nec.com at Tue Apr 3 18:45:24 2001
Starting the Scheduler.....
perf: Queue-Handling 0.31
perf: Context Switch 1.29
perf: Routine Call 0.01
perf: Protected Routine Call 0.08
perf: Queue-Handling 0.31
perf: Context Switch 1.29
perf: Routine Call 0.01
. . .
```

and every 15 minutes the uptime process runs and produces an extra message:

```
uptime: 0:15
```

The exact performance figures vary with the exact model of CPU on which Perf is running and the behaviour of the instruction cache.

Understanding the impact of these low-level timings is not trivial. The results themselves may vary according to the line cache algorithms and the exact locations at which instructions are assembled. ROME gives some control over this by allowing external control over the order in which modules are included in the system, and the start address of the text. A full analysis of caching and code-placement is beyond the scope of this tutorial. The main insight to be gained from the performance measurements is for cross-comparisons between different boards for potential applications.

It may be possible to trade off implementations for particular achitectures; for example to remove a Queue Handler on a system where routine calls have a high overhead. In theory this could be achieved through module configuration and option directives, but this has not yet been explored for version 1.0 of ROME.

5 Tutorial IV: timer_pc

This tutorial introduces the facilities in ROME for handling devices, using the counter/timer modules for the Intel system. This also serves to ‘close the loop’ on the previous example by showing the processing in the timer shared library.

In many operating systems, the code for handling devices runs in a very special environment which adds to the complexity of producing a reliable implementation. In ROME, a device driver is just a module, with a process and possibly a queue handler, as described above. There are two requirements placed on device drivers that fall outside the scope of the interfaces described so far. This tutorial introduces these two concepts.

The first is the requirement to run code associated with initialising the device. Usually this code must run before the normal system scheduler is started, to set up the registers and low-level operation of the device.

In ROME this is the other routine name (the *init* routine) that can be given on the RTB process dialog. The init routine is called from the core during system startup before the scheduler runs. The routine runs with interrupts disabled, and cannot use the message-passing routines.

The second requirement is to install a routine for handling device interrupts. This is done by calling the *rome_add_handler* core routine, giving the interrupt number and the routine to be used. Usually this call is placed in the init routine, so that the handler is in place when the scheduler is started, at which point interrupts are enabled.

5.1 Timer Functionality

Most PC architectures implement a standard built-in counter/timer. A special-purpose register may be loaded with a timer value to cause an interrupt at a later time. The timer initialisation routine uses this register to set up an interrupt every millisecond. The module also maintains a queue of timeout requests, linked in timeout order. At each 'tick' the counter at the head of queue is decremented by one. When the counter reaches zero, the reply is sent. The operation of adding a message to the queue is handled within the timer process Queue Handler and the reply is sent from the Interrupt Handler.

The user interface to the timer process hides the details of the message format behind the *timeout* and *timer_tmhandler* routines, as used in the previous tutorial. These routines are provided as part of the shared-library enviroment, like the 'C' runtimes. The usual practice in ROME systems is to provide a routine to construct a message on behalf of a caller. This also extends to the dataflow messages described in the following chapters.

The module also uses a set of *Option* fields for testing the timing code. Since this is not the focus of this tutorial, the testing code is not included in the listings here, but is available in the full sources.

5.2 The Target File

The value that must be programmed in to the PC timer register depends on the exact processor implementation and the specific hardware configuration of the target system. One possible means of specifying this system-dependent value would be as an *Option* line in *RTB*. However, such values do not (usually) change between applications built for the same hardware. All the various hardware dependencies for a system are instead gathered together into a single description for that hardware. These are the files that are referenced by the *Target* component in the build files.

The target file contains the rest of the information needed to complete the full configuration picture for the examples presented so far. It contains the information used to set up the *Makefile* rules, to select the correct cross-compiler and assembler and to generate the appropriate *ld* output format. It also contains data which appear in the *Hardware.h* file in the *include* directory.

Within the master tree is the *Targets* directory which contains the supported system configurations. In here is the *SPB450* file which describes the PC environment used in this tutorial. Within the system file are the definitions for this configuration:

```
#define TIMER_TICKS2SEC  1000
#define HZ               KC_TICKS2SEC
#define TIMER_TIX        1193
#define TIMER_VEC_TICK   ICU_IRQ0
#define TIMER_CLEAR_INT  CPU_IOWR1(0x20, 0x20)
```

This also defines the *HZ* variable used to convert from seconds to clock ticks at the *timeout* interface.

5.3 The RTB File

The timer_pc module comprises two source files, one for the low-level driver and one for the interface library. The interface is specified through an external header file. In the 'C Files' part of the project are two source files, *timer.c* and *timerlib.c*. In the 'H Files' part is *timerlib.h*, which is also marked as 'exported' so that it appears in the project's *include* directory.

The *TIMEOUT* inter-process message is defined in the Timer MessageSet with a single integer parameter:

```
Message TIMEOUT
{
    int ticks;
}
```

The process description contains entries for the initialisation routine and queue handler as well as the main process.

5.4 The timerlib shared library

The purpose of the timerlib library is to hide the details of the timer message interface from application programs and to implement the 'callout' functions required in many applications. The implementation of the *timeout* routine is less straightforward than might appear because of the need to store the context information (the pointer to the routine and its argument) to allow the reply to be correlated with the original request.

5.4.1 Local Data

The library uses a private data structure to store context information:

```
typedef struct
{
    ROME_MESSAGE      m;          message sent to timer
    int               flag;       if still active
    void(*func)(caddr_t);        completion routine
    caddr_t           funcarg;    argument to routine
} TMOUT;
```

The structure contains the actual instance of the *ROME_MESSAGE* which is sent to the timer process.

5.4.2 The timeout routine

The *timeout* routine used above is actually a macro interface to the *timeout_proc* routine. This routine allocates a *TMOUT* structure, initialises the fields and sends the message to the timer process.

```
int timeout_proc(ROME_PROCESS *who, void (*func)(caddr_t),
                 caddr_t se, int hz)
{
    TMOUT *tm = (TMOUT *)rome_alloc(sizeof(TMOUT), 't', 1);
    ROME_T_TIMEOUT *to = RCAST(ROME_T_TIMEOUT, tm->m);
```

The first line of the routine uses the core *rome_alloc* routine to allocate and clear a block of memory for the context. *rome_alloc* halts the system if no memory is available, so the routine here has no need to check for *NULL* pointers. The second parameter is an allocation identifier, which can be displayed from the debugger, indicating from where this allocation request originated.

The second line introduces a new macro from the ROME core. The *RCAST* macro takes a message type as its first argument, and a *ROME_MESSAGE* structure as its second argument and returns a pointer to the argument fields of a message, cast into the correct type for the message. The *RCASTP* macro performs exactly the same function except that its second argument is a pointer to a *ROME_MESSAGE* structure.

```
    tm->flag = 1;
    tm->func = func;
    tm->funcarg = se;
    to->ticks = hz;
    tm->m.src_context = tm;
    tm->m.dest = rome_find_queue("timer");
    tm->m.src = who;
    tm->m.opcode = ROME_M_TIMEOUT;
    tm->m.priority = who->prio;
    rome_send_message(&tm->m);
    return (int)tm;
}
```

By default, processes generate messages at their current scheduling priority. The message priority is initialised from the process' current state. This allows the timer library to run as a shared library possibly with concurrent execution in multiple contexts. The macro is actually:

```
#define timeout(a, b, c) timeout_proc(rome_this_ptr, a, b, c)
```

The ROME core maintains the *rome_this_ptr* variable and the appropriate fields are available as contexts are switched. The reason for the *timeout* and *timeout_proc* level of indirection is to allow certain routines to bypass the current-process pointer, for example to create timers on behalf of other processes. This is needed, for example, to create a timer within an interrupt handler. All of the variables for the routine are held on the process stack, so the routine is truly sharable. No interlocks are needed for any of this routine.

The *hz* argument is used to initialise the *ticks* parameter of the message, which is sent to the "timer" process. For this routine to function correctly, the timer must be linked into the system with the name "timer", but the operation of the library interface does not depend on the implementation of the actual code used for the system timer.

ROME messages contain fields that can be used by the sender and the recipient of the message to store information. Here, the *src_context* field is used (since this is the source of the message), to point to the timeout structure.

5.4.3 The untimeout routine

The *untimeout* routine is used to cancel a timer. This is less obvious than it might appear, as the timer message may be in a number of places: it may still be on the queue of the timer process, or it may be queued back to the originator. The ROME implementation takes the 'easy way out', by allowing the timeout to occur, but not calling the user's routine. It signals this by setting the *flag* field to zero:

```
void untimeout(int cx)
{
    ((TMOUT *)cx)->flag = 0;
}
```

This also allows the allocated memory to be freed in a clean way.

5.4.4 Sleep

The *sleep* routine suspends a process for a number of timer intervals, using a timeout message. That is, it prevents the process from accepting any messages, or otherwise occupying the CPU, during that period. In order to do that, it must wait until the reply to the timeout comes back to the process:

```
void sleep(int hz)
{
    ROME_MESSAGE m;
    ROME_T_TIMEOUT *to = RCAST(ROME_T_TIMEOUT, m);

    to->ticks = hz;
    m.dest = rome_find_queue("timer");
    m.src = rome_this_ptr;
    m.opcode = ROME_M_TIMEOUT;
    m.priority = rome_this_ptr->prio;

    rome_send_message(&m);
    rome_await_message(&m, 0);
}
```

Since it only makes sense to sleep in a process context, the *rome_this_ptr* variable can be used directly. Most of this code is similar to the internal *timeout_proc* routine, except that no *TMOUT* context is needed. The main difference is the final line, which is an explicit call to *rome_await_message.* By giving a pointer to a message as the first parameter to *rome_await_message*, the core scheduler suspends the process until that particular message is placed on the process' message queue. That message is then returned as the result of that particular call, irrespective of any other messages which may be on the queue. Because the message is local to the *sleep* routine, it can be allocated on the process' local stack.

The approach of using local messages and waiting for the replies is used in the interface library routines to the main dataflow messages, explained below. In this way, it is possible to mix event-driven messages with blocking calls.

5.4.5 The timer_tmhandler routine

The *timer_tmhandler* code completes the cycle by calling the user's routine and releasing the resources:

```
void timer_tmhandler(ROME_MESSAGE *m)
{
    TMOUT *tm = (TMOUT *)m->src_context;
    if  (tm->flag)
    {
        (tm->func)(tm->funcarg);
    }
    rome_free(tm);
}
```

The source context field in the message is used to recover the pointer to the *TMOUT* structure, and the *flag* field is checked before the user's routine is called. Finally, *rome_free*, which is a core routine, returns the context memory to the (global) free memory pool.

5.5 The Clock routines

The other half of the timer support is the set of routines to interface with the hardware.

5.5.1 The timer init routine

The first routine to be called is the initialisation routine:

```
void timer_init(void)
{
    tick_counter = 0;
    timerq = NULL;
    rome_add_handler(TIMER_VEC_TICK, timer_isr);

    CPU_IOWR1(TIMER_CONTROL, 0x36);
    CPU_IOWR1(TIMER_STATUS0, TIMER_TIX & 255);
    CPU_IOWR1(TIMER_STATUS0, TIMER_TIX >> 8);
}
```

The routine initialises the variables used by the interrupt routine and registers the handler for the tick timer using the core routine *rome_add_handler*. The interrupt vector number comes from the hardware description in the target file. The routine also sets the values for the timer registers. Accesses to machine-dependent architectural features (like IOSpace) are handled by the CPU plug-in through macros. In this case, *CPU_IOWR1* writes a single byte to a particular IO location. More details of these macros is given in the CPU plug-in documentation, and the ROME Porting Guide

The *timerq* variable is used to hold the queue of timer requests as a linked list of messages.

5.5.2 The Queue Handler

This routine controls the queue of timer requests:

```
int timer_qhandler(ROME_MESSAGE *mptr)
{
    switch (mptr->opcode)
    {
    case ROME_M_TIMEOUT :
    {
        ROME_T_TIMEOUT *to = RCASTP(ROME_T_TIMEOUT, m);

        mptr->link = 0;
        if  (to->ticks <= 0)
        {
            rome_auto_reply(mptr);
        }
```

This first check filters out any messages with timeout requests in the past, or with zero timeout values (as might arise if the interval is calculated by the originator based on processing activity).

```
        if (timerq)
        {
            ROME_MESSAGE *mp = timerq;
            ROME_MESSAGE *pp = timerq;
            ROME_T_TIMEOUT *tp = RCASTP(ROME_T_TIMEOUT, mp);

            while (tp->ticks < to->ticks)
            {
                to->ticks -= tp->ticks;
                if  (mp->link)
                {
                    pp = mp;
                    mp = mp->link;
                    tp = RCASTP(ROME_T_TIMEOUT, mp);
                }
                else
                {
                    mp->link = mptr;
                    return ROME_HANDLED;
                }
            }
```

Timeout messages are chained in a linked list in timeout order, with the *ticks* value representing the difference between the previous timer and the current one. This means that the shortest timeout is always at the head of the queue, also that multiple timers for the same tick will appear with zero *ticks* in the chain. Since linking in message chains is a common operation (also used within the core), each message contains an explicit *link* field intended to point to another message. The first loop scans down the queue for the right place to put the new message. The message is added to the end of the queue if there is no message with a longer timeout interval.

```
            if  (mp == timerq)
            {
                mptr->link = timerq;
                timerq = mptr;
            }
            else
            {
                pp->link = mptr;
                mptr->link = mp;
            }
            tp->ticks -= to->ticks;
        }
        else
        {
            timerq = mptr;
        }
        return ROME_HANDLED;
    }
    break;
```

The first test handles adding the new message to the head of the timer queue, or adding to the middle of the queue (inbetween *pp* and *pp->link*). The final else clause handles the case of an empty initial queue.

In all cases the result returned is *ROME_HANDLED* to indicate to the core that the message has been processed in the queue handler

```
        }
        return ROME_NOT_HANDLED;
    }
```

Finally, all other messages are returned with the 'not handled' indication, meaning they will be passed on to the main process (below).

5.5.3 The Main Process

The main process code is very simple:

```
void timer_process(void)
{
    while (1==1)
    {
        ROME_MESSAGE *m = rome_await_message(0, 0);

        rome_kprintf("timer message %x??\n", m);
    }
}
```

The only message that is supposed to be sent to the timer process is the *TIMEOUT* message. So all messages are, in theory, handled by the Queue Handler, and the main process should never be woken from its wait. The main purpose of the process is to provide a reference for the "timer" process name. The print serves to alert a user to a probable misconfiguration of the system, as it should never appear.

5.5.4 The interrupt handler

This routine is called from within the CPU plug-in in response to the timer interrupt:

```
void timer_isr(int ino)
{
    if  (timerq)
    {
        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);
        }
    }
    if  (++tick_counter == KC_TICKS2SEC)
    {
        tick_counter = 0;
```

```
            secs++;
        }
        TIMER_CLEAR_INT;
    }
```

If there is a timeout message on the queue, the count remaining for that message is decremented. When it reaches zero, all messages for that tick are returned as replies. When the internal *tick_counter* has counted for one second it is reset and the internal *secs* counter is incremented. Finally, the *TIMER_CLEAR_INT* macro, defined in the target file, is used to clear the pending timer interrupt. Note that both the Queue Handler and the Interrupt Handler are manipulating the *timerq* variable. It is essential that neither interferes with the operation of the other. In this case, the automatic protection offered to the Queue Handler of running as a critical section is used to ensure that the interrupt handler cannot run during the update. Interrupt Handlers also run as critical sections, so resource interlocking is achieved without further explicit procedure calls.

5.5.5 Variables

```
static int secs = 0;
static int tick_counter;
ROME_MESSAGE *timerq;
```

The timer queue variable is local to this file, but exporting the symbol helps when debugging timing problems (see also the debug part of chapter 7).

5.6 Modifying the Clock Driver

The previous sections have described the functionality of the clock process as used in the previous two tutorials. No new builds are needed for this description.

The second purpose of this tutorial is to modify the clock driver to print a '+' sign every second. This will be produced from the clock interrupt handler, which runs in an environment where *printf* cannot be used. ROME provides a special routine to generate output in these environments, called *rome_kprintf*. The routine runs as a critical section and writes directly to the UART.

5.6.1 Making the source file writable

If the sources for modules have been checked out from a CVS repository, then a module must be locked before it can be updated. This is done in *RTB* by selecting the module in the project summary listing and selecting the 'lock' option from the popup menu (right-click on the mouse). Once this is done, the source files in the *Modules* directory will be writable. It is not necessary to rebuild the project Makefile tree.

5.6.2 Changing the source

Using the *Perf* project, lock the *timer* module and edit the file *timer_pc/timer.c* changing the following lines in the *timer_isr* routine from:

```
if (++tick_counter == KC_TICKS2SEC)
{
    tick_counter = 0;
    secs++;
}
```

to:

```
if (++tick_counter == KC_TICKS2SEC)
{
    rome_kprintf("+");
    tick_counter = 0;
    secs++;
}
```

then save the changes and run *make install* as before.

5.6.3 Running the image

When the image is run, the performance values are printed with ten '+' signs between them.

```
++++++++++perf: Queue-Handling 0.31
++++++++++perf: Context Switch 1.29
++++++++++perf: Routine Call 0.01
++++++++++perf: Protected Routine Call 0.08
++++++++++perf: Queue-Handling 0.31
```

depending on the exact system timing, the '+' characters may appear in the middle of the usual perf output, since the *rome_kprintf* routine runs at absolute priority and bypasses the UART queueing mechanism which prevents interleaved output. The implementation of this mechanism is described in tutorial V.

6 Tutorial V: UART16550

The main focus of this tutorial is to introduce the standard ROME dataflow messages and demonstrate how they are handled within a device.

The applications so far (such as *echo*) have been unusual for an embedded processor application in that the dataflows terminate within the machine. A more usual model would have data being received by one device, being manipulated by one or more processes within the machine and finally leaving the machine through another (or even the same) device. If every device driver defined its own set of messages for data transfer there would need to be a $N \times N$ translation operations, possibly requiring data copies, and there would be no standard interworking between modules.

Instead, ROME defines a standard set of messages to be used for data movement. The design is intended to address a number of issues:

(a) The goal is to move data from input device to output device without any unnecessary copies (ideally without any copies).

(b) For some input devices, notably networked systems, the target of a data block cannot (may not) be known until the block has been received into memory.

(c) Some devices may require buffers to be allocated in a special way, for example within a given region of memory, or not spanning particular addressing boundaries.

(d) In contrast, some devices, for example disk systems, may operate in a completely deterministic manner. Not only may it not matter where buffers are located, but the destination of a buffer can be determined before a request is issued.

(e) Network protocols require the addition and removal of headers (and trailers) from data packets.

The approach used in ROME is based around the ideas of STREAMS (See, for example the *SunOS 5.3 STREAMS Programmer's Guide*, (Sun Microsystems Inc., 1993)), but intended to be used throughout the system, not just within the 'kernel'.

6.1 mblks

An *mblk*, or message block, is the basic representation of data in transit through the system. An area of memory used as a data buffer is represented by three variables:

```
uchar *b_base;     original base of buffer
uchar *b_lim;      absolute end of buffer
uint   b_type;     buffer type;
```

where *b_base* points to the start of the buffer and *b_lim* points to the end of the buffer. The *b_type* field is used in protocol processing to identify the type of data in the buffer.

The size of the buffer is *b_lim*-*b_base* bytes. Note that these are only *pointers* to a buffer, the structure does not itself reserve that area. It also does not define how the area within the buffer is used.

The *mblk* structure represents data within a buffer as follows:

```
uchar *b_rptr;     current read pointer
uchar *b_wptr;     current write pointer
```

The *b_rptr* and *b_wptr* fields define the area of the data block containing valid data. *b_rptr*, the read pointer, defines the point at which a process should start reading data (i.e. the beginning of the buffer). *b_wptr*, the write pointer, defines the point at which a process should append data (i.e. the end of the buffer). The valid data lie between these two pointers and are of length *b_wptr*-*b_rptr* bytes.

The read pointer need not coincide with the start of the data block, and the write pointer need not point to the end of the data block. By setting the read and write pointers for a new message block some way into the data block it is possible to 'reserve' space at the start of a block. This aspect will be the focus of the next tutorial.

A single unit of data may span more than one buffer. Multiple mblks can be chained together using the continuation field, *b_cont*:

```
struct mblk_t *b_cont;
```

To complete the mblk data definition, two fields are available for per-buffer 'immediate' data:

```
uint b_immed;
uint b_immed1;
```

An example of the use of an immediate data field will be given below. The basic approach to these data structures is the same as the examples in the timer and performance code. The same portion of a ROME message that is used to contain the *ticks* value for the *TIMEOUT* message can also be used to contain these *mblk* fields. The values can then be manipulated as the message passes through the system. However, since most messages use the *mblk* fields, rather than requiring *RCAST* or *RCASTP* macros to access the fields, they can be accessed directly within the message. In fact, to simplify the porting of STREAMS-based drivers to the ROME environment, the ROME_MESSAGE and *mblk_t* data types are one and the same!

This has given enough background to allow the full *ROME_MESSAGE* structure to be given as it appears in *rome.h*. The use of the *dest*, *source*, *opcode*, *priority*, *dest_context* and *src_context* fields have been covered in the previous tutorials. The *link* field was used in the Clock driver, but is also used within the core for chaining messages to process queues. Each message also contains an error code field. The remainder of the fields are used to contain an *mblk*:

```
#define b_cont     residue.imblk.ib_cont
#define b_rptr     residue.imblk.ib_rptr
#define b_wptr     residue.imblk.ib_wptr
#define b_base     residue.imblk.db_base
#define b_lim      residue.imblk.db_lim
#define b_type     residue.imblk.db_type
#define b_immed    residue.imblk.immed
#define b_immed1   residue.imblk.immed1
#define m_args     residue.args.avec

#define RCAST(_t, _m)    (_t *)((_m).m_args)
#define RCASTP(_t, _m)   (_t *)((_m)->m_args)

typedef struct _rome_message
{
    struct _rome_message *link;             to next on queue
    struct _rome_process *dest;             destination queue
    struct _rome_process *src;              source queue
    int m_errno;                            see errno.h
    uint opcode;                            see messages.h
    uint priority;                          this message priority
    ptr dest_context;                       opaque field for dest use
    ptr src_context;                        opaque field for src use
    union
    {
        struct
        {
            struct _rome_message *ib_cont;  continuation block
            uchar *ib_rptr;                 current read pointer
            uchar *ib_wptr;                 current write pointer
            uchar *db_base;                 original base of buffer
            uchar *db_lim;                  absolute end of buffer
            uint db_type;                   buffer type
```

```
            uint immed;                     immediate data
            uint immed1;                    immediate data 1
        }imblk;
        struct
        {
            uint avec[8];                   depends on code
        }args;
    }residue;
}ROME_MESSAGE, mblk_t;
```

The structure contains the linkage fields and context fields followed by a 32-byte argument area *residue*, which can be viewed either as an 8-word array *args* or as *imblk*. The conventional names for the mblk fields are defined symbols addressing these fields.

6.2 The message interface

This section describes the messages that are defined as part of the standard ROME interface. Three messages, OPEN, CLOSE and FLUSH, handle control operations. Six messages, FETMBLK, GETMBLK, NEWMBLK, OUTMBLK, PUTMBLK and RETMBLK are used for moving buffers of data around the system.

6.2.1 Control Messages

ROME_M_OPEN

In order to use ROME dataflows between processes, the destination process must be located and any state within that process established for the particular flow. Locating processes, as with the timer library, is done with the core routine *rome_find_process*. Initialising the state in the destination is done by sending an *OPEN* message to the destination. There may be many separate flows between any two pairs of processes, each with its own internal state, and there needs to be some way of associating messages with flows. When a process receives an *OPEN* message it has the option of storing a 'context' value in the reply to identify the flow. This 32bit value is returned in the *dest_context* field of the reply. The same value will be set on all subsequent messages on that flow. This is handled by the runtime library which implements the *FILE* abstraction on top of the basic messages.

The *OPEN* message has its own parameter area:

```
typedef struct
{
    struct _rome_url openurl;
    int mode;
}ROME_T_OPEN;
```

The first parameter is a pointer to a ROME URL data structure used to identify the destination, the second is a bitfield of open types. Currently this bitfield contains the *BINARY, READ* and *WRITE* bits derived from the 'C' *fopen* routine.

The full *openurl* is an expanded form of the Uniform Resource Locator (URL) where the *scheme* part is used to identify a ROME process (see, for example 'Uniform Resource Locators', RFC 1738). The ROME URL will be described in more detail in the next chapter.

ROME_M_CLOSE

The *CLOSE* message is used to inform a process that it can terminate a flow and release all the flow resources. The only information passed on the *CLOSE* message is the *dest_context* field.

ROME_M_FLUSH

Any process in the system is allowed to buffer data. It may even return a reply to the originator indicating that the data has been 'handled' before a transaction has completed (for example TCP may reply before the other side has acknowledged the data). The *FLUSH* message synchronises a dataflow. The reply to a *FLUSH* should not be generated until all previous data have been transmitted and acknowledged (for reliable service).

6.2.2 Moving Data with Messages

Having established the context for a flow, data can be moved along it. The main issue is the management of the memory containing these data, as a system resource. That is, who allocates this memory; who is currently 'responsible' for it; who has the ability, or the requirement, to free it when it is no longer needed; and who determines that the data are no longer needed?

In the context of ROME, these are not trivial questions. The dataflow mechanism is designed to allow buffers, represented by mblks, to be passed between processes, and the goal is a true zero-copy architecture. This implies that the process that originally filled a buffer is unlikely to be the one that decides the data in it are no longer needed. The process that does wish to free the buffer may have no direct access to, or knowledge of, the process that created it. Also, some devices place constraints on their buffers – they may have special alignment or boundary requirements (such as lying within a 64k addressing page) or they may live on special memory (for example uncached memory for DMA-capable devices).

For some devices, the ultimate destination process of a buffer of data may not be known until that data is completely received (for example ethernet frames). In contrast some devices, such as disk drives, only deliver data in response to explicit requests and often into buffers associated with those requests.

The ROME dataflow model attempts to solve these issues in as simple a way as possible, using only six data movement messages.

ROME_M_FETMBLK

The *FETMBLK* message is a request to fill an mblk with data. The buffer for the data is supplied at the time to message is issued (i.e. it is owned by the originator). The recipient should fill in data starting at the *b_wptr* value and update that field to reflect the new length. The *FETMBLK* message is typically sent to a process that does deterministic data movement, for example a disk device.

ROME_M_GETMBLK

The *GETMBLK* message is a request for a block of data. The buffer for the data is allocated by the recipient, and must be returned to it. The reply contains the *mblk* fields initialised to point to the data buffer. Note that a single buffer of data may span multiple blocks, these are chained through the *b_cont* field. The buffer is returned to the owner with the *RETMBLK* message.

ROME_M_NEWMBLK

The *NEWMBLK* message requests an empty buffer into which the sender can write data. The length of the buffer is passed in the *b_wptr* field of the original message. The reply will have the *b_wptr* field set to

point to the first writable byte of the data buffer. Note that protocol header processing requirements may mean that this is not the *b_base* location. The buffer must be returned to the allocator with a *PUTMBLK* message.

ROME_M_OUTMBLK

The *OUTMBLK* message requests the recipient to transmit the associated data block in the 'downstream' direction (towards the outside world). When transmission is complete, the data buffer is returned, unchanged to the originator, i.e. it is a 'write and keep' operation.

ROME_M_PUTMBLK

The *PUTMBLK* message requests the recipient to transmit the associated data block in the 'downstream' direction (towards the outside world). When transmission is complete, the data buffer is freed. This requires that the recipient originally allocated the buffer and yielded it with a *NEWMBLK* message. To transmit data through a process which does not own the buffer, the *OUTMBLK* message must be used.

If the data length in a *PUTMBLK* message is zero (i.e. *b_wptr == b_rptr*) no data are transmitted and the buffer is freed immediately. This may be used to return a buffer that is no longer required, or to return a buffer for data previously transmitted by an *OUTMBLK* operation.

ROME_M_RETMBLK

The *RETMBLK* message is used to return a data buffer that was previously passed upstream in a *GETMBLK* reply. Note that *RETMBLK* returns buffers from *GETMBLK* and *PUTMBLK* returns buffers from *NEWMBLK*. This allows different buffer strategies to be implemented for transmission buffers and reception buffers.

6.3 Application to the UART16550 driver

The UART16550 driver provides an interface to any UART implementation based on the NS16550 register map. It provides characters, one-at-a-time, to satisfy input requests, and outputs characters from *PUTMBLK* or *OUTMBLK* requests. The UART does not itself perform character echoing, or interpret format effectors (for example backspace and delete). It is also expected that only one process will open a data path to the UART directly. In these systems that is the Console module.

Since the UART is a terminating device which does not itself originate messages, it only has to process the request forms of the messages. A process which sits 'mid-stream' must handle requests from upstream and replies from downstream. This is covered in the next tutorial.

6.4 The Target File

Because the code is designed to work with more than one implementation, various parameters are used in the target file to configure the sources at compilation time:

```
#define SERIAL_UART16550_BASE0        0x2f8
#define SERIAL_UART16550_BASEX        0x3f8
#define SERIAL_UART16550_VEC_INT0     ICU_IRQ4
#define SERIAL_UART16550_VEC_INT1     ICU_IRQ3
#define SERIAL_UART16550_REG_SPACING  1
#define SERIAL_UART16550_MCR          (MCR_OUT2 | MCR_DTR)
```

```
#define SERIAL_UART16550_ADD_HANDLER(_a, _b) rome_add_handler(_a, _b)
#define SERIAL_UART16550_CLEAR_INT0   CPU_IOWR1(0x20, 0x20)
#define SERIAL_UART16550_CLEAR_INT1   CPU_IOWR1(0x20, 0x20)
```

These definitions encapsulate the variations that are found between different boards that use this chipset. For example, in some architectures, the UART registers are spaced one byte apart, which in other they are space four bytes apart (depending on the address decode logic). This is set by the *REG_SPACING* parameter.

6.5 Source Code

The UART driver contains the same basic structure as that clock driver of the previous tutorial, though the operation of the individual routines are rather more complicated, as many more message types are handled. As the main focus of this tutorial is on message handling, some of the device-specific code (for example handling multiple UARTs) will be omitted. All the device code uses a set of macros to encapsulate the accesses to the UART registers. These macros in turn use the CPU-specific IOSpace access macros defined by the plug in, for example:

```
#define WR(_r, _v)     CPU_IOWR1(port->uart_port + (_r), (_v))
#define RD(_r)         CPU_IORD1(port->uart_port + (_r))
```

for writing and reading a byte to the *_r* register of the current port.

The structure of this tutorial differs from the previous ones. Rather than go through the Queue Handler, Process and Interrupt Handler routines line-by-line, the following sections focus on how each message is handled within the module.

6.5.1 The init routine

A short routine ensures a clean state for the rest of the code:

```
void serial_init(void)
{
    SERIAL_PORT *port;

    port = &ports[0];
    port->uart_port = (volatile uchar *)SERIAL_UART16550_BASE0;
    SERIAL_UART16550_ADD_HANDLER(SERIAL_UART16550_VEC_INT0, serial_isr);
    port->readq.head = port->readq.tail = NULL;
    port->writeq.head = port->writeq.tail = NULL;
    port->todo = '\0';
    WR(IER, IER_RxIE | IER_RLS);
}
```

The interrupt handler uses a queue of read requests for input, and a queue of write requests for output. Input messages are returned in the order they are received and output is sent to the UART in the order in which it is generated. These both require FIFO queues, which are added to at the tail end and from which elements are removed at the head end. This is a sufficiently common requirement for messages that ROME supports this queue by a combination of macros and shared-library routines. The routines require

only that the head and tail pointers of a queue be NULL when the queue is empty. This is ensured as part of the module initialisation:

The final line of the routine enables the receive interrupt for UART0.

6.5.2 The Interrupt Handler

There is some code in the interrupt handler that is not directly related to message handling.

```
static void serial_isr_internal(int uix)
{
    SERIAL_PORT *port = &ports[uix];
    uchar ipend = RD(IIR);

#ifdef ROME_TRACE_INTERRUPTS
    rome_add_trace((ptr)(uint)ipend, ROME_TT_STARTINT, (ptr)uix);
#endif

    ipend &= IIR_MASK;

    if  (ipend == IIR_RDA || ipend == IIR_TIMEOUT)
    {
        . . . input handling
    }
    if  (ipend == IIR_THRE)
    {
        . . . output handling
    }

#ifdef ROME_TRACE_INTERRUPTS
    rome_add_trace(0, ROME_TT_ENDINT, (ptr)uix);
#endif
}
```

The calls to the *rome_add_trace* routine record the entry and exit to the interrupt handler, if the *ROME_TRACE_INTERR* option is set. This debugging aid is described later in this chapter. The interrupt handler reads the status register of the UART which generated the interrupt and processes pending input characters and pending output characters.

6.5.3 The OPEN Message

Queue Handler

```
ROME_T_OPEN *oopen = RCASTP(ROME_T_OPEN, mptr);
uint uix = open->openurl->port;

mptr->dest_context = (ptr)uix;
mptr->m_errno = 0;
rome_auto_reply(mptr);
```

The UART code is unusual in handling the *OPEN* message in the Queue Handler. It does this to prevent the Console process from blocking when it opens the dataflow to the default output device. Having Console

complete process startup without blocking prevents circular message lockup with processes which are trying to open standard streams. It is assumed that only one process ever opens a file directly to each UART port, so there is no need to record any special per-process context information. The routine returns the index of the open UART as the source context in the reply. The UART index is derived from the URL structures, which is explained in the next chapter. In all the following messages, the port index is passed in through the message, and the context is recovered using the following declaration:

```
SERIAL_PORT *port = &ports[(int)mptr->dest_context];
```

this line is implied at the start of all the following code fragments.

6.5.4 The CLOSE Message

Main Process

```
ROME_MESSAGE *xx = rome_remhead(&port->readq);

while (xx)
{
    xx->m_errno = ECANCELED;
    rome_reply(xx);
    xx = rome_remhead(&port->readq);
}
mptr->m_errno = 0;
rome_reply(mptr);
```

In processing the *CLOSE* request, any messages being held by the UART for that flow must be returned to the originator with an error indication. This is handled within the process. Note that all the replies are queued to the originator before the reply to the *CLOSE* itself.

6.5.5 The FLUSH Message

Main Process

```
if (port->writeq.head == 0)
{
    rome_reply(mptr);
}
else
{
    int old = rome_start_critical();

    rome_addtail(&port->writeq, mptr);
    rome_end_critical(old);
}
```

The requirement for *FLUSH* is that the message is returned only after all preceding output has completed. If there is no output pending, the message is returned immediately; otherwise the message is added to the tail of the output queue. Note that in this case, running in the process context, the routine must take out an interlock in order to update the port *writeq* variable.

Interrupt Handler

```
while (port->writeq.head && port->writeq.head->opcode == ROME_M_FLUSH)
{
    mp = rome_remhead(&port->writeq);
    rome_reply(mp);
}
```

Once a *FLUSH* message arrives at the head of the output queue (because all earlier buffers have been transmitted) it can be returned to the originator.

6.5.6 The FETMBLK Message

Queue Handler

```
rome_addtail(&port->readq, mptr);
return ROME_HANDLED;
```

Incoming *FETMBLK* requests are added to the tail of the port's read queue.

Interrupt Handler

See the description for *GETMBLK*, below.

6.5.7 The GETMBLK Message

Queue Handler

```
mblk_setup(mptr, (uchar *)&mptr->b_immed, 4);
rome_addtail(&port->readq, mptr);
return ROME_HANDLED;
```

The GETMBLK message, which expects the UART to provide the buffer, uses instead the *b_immed* area of the mblk, since it only ever returns one character. The support library routine *mblk_setup* initialises all the fields of the message block for a given buffer. The handler then adds the message to the tail of the queue of reads for the UART.

Interrupt Handler

```
char data = RD[RXD];

if  ((mp = rome_remhead(&port->readq)))
{
    *mp->b_wptr++ = data;
    rome_reply(mp);
}
```

This code lies within the 'input handling' section of the interrupt routine. The available input character is removed from the UART's FIFO. If there is a request on the read queue, the character is passed up in the reply. The write pointer is incremented, indicating one byte in the buffer. Note that it does not matter whether this is a *FETMBLK* message or a *GETMBLK* message at this point.

6.5.8 The NEWMBLK Message

Queue Handler

```
OUTPUT_BUFF *op = freeq;

if (op)
{
    freeq = op->link;
}
else
{
    op = (OUTPUT_BUFF *)rome_alloc(sizeof(OUTPUT_BUFF), 4, 1);
}
mblk_setup(mptr, op->input, 128);
rome_auto_reply(mptr);
```

Although input is handled character-by-character, output may be presented in larger buffers, typically containing a full line. Available output buffers are held in a linked list. If there is a buffer, it is used, else a new one is allocated. The *mblk_setup* routine initialises all the fields, as with *GETMBLK* above.

6.5.9 The OUTMBLK Message

Queue Handler

```
if  (port->writeq.head == NULL)
{
    port->currtxm = mptr;
    port->currtxp = mptr->b_rptr;
    SET(IER, IER_TxIE);
}
rome_addtail(&port->writeq, mptr);
return ROME_HANDLED;
```

If the UART transmit queue is empty, the UART must be re-started, and the ‘transmit available’ interrupt enabled. The handler uses the *currtxm* and *currtxp* pointers to control individual character transmissions. Note that, since the Queue Handler runs as a critical section, updating these variables is automatically interlocked against the interrupt service routine. In all cases the message is added to the end of the writer queue for the port.

Interrupt Handler

See the description for *PUTMBLK*, below.

6.5.10 The PUTMBLK Message

Queue Handler

```
if (mptr->b_rptr == mptr->b_wptr)
{
     OUTPUT_BUFF *op = (OUTPUT_BUFF *)mptr->b_base;

     op->link = freeq;
     freeq = op;
     rome_auto_reply(mptr);
}
(the rest of the code is the same as OUTMBLK above)
```

For *PUTMBLK*, the routine must check for the zero-length indication used to return unwanted buffers. In this case, the buffer is added to the free queue. Since there is no ordering required for these buffers, a simple head-linked list suffices. Otherwise, the processing is the same as for *OUTMBLK* above.

Interrupt Handler

The following code is contained in the 'output handling' part of the interrupt service routine.

```
if  (port->todo)
{
     WR(TXD, port->todo);
     todo = '\0';
}
```

The *todo* variable holds a single character which is next to be output. This is used to convert 'newline' to 'newline' + 'carriage-return' for standard terminals, without changing the contents of the supplied output buffer.

```
else if (port->currtxm)
{
     if (*port->currtxp == '\n')
     {
          todo = '\r';
     }
     WR(TXD, *port->currtxp++);
```

if there is a character awaiting output, these lines handles the CRLF translation, and output the next character from the buffer.

```
     while (port->currtxm && port->currtxp == port->currtxm->b_wptr)
     {
          port->currtxm = port->currtxm->b_cont;
          if (port->currtxm)
          {
               port->currtxp = port->currtxm->b_rptr;
          }
     }
```

A single output request may contain a linked list of buffers, each of which is to be printed. This code moves the internal pointers to the next non-empty character position in the chain, and leaves *currtxm* as *NULL* if the current message has been completely transmitted.

```
        if (port->currtxm == NULL)
        {
            ROME_MESSAGE *mp = rome_remhead(&port->writeq);

            if  (mp->opcode == ROME_M_PUTMBLK)
            {
                OUTPUT_BUFF *ob = (OUTPUT_BUFF *)mp->b_base;

                ob->link = freeq;
                freeq = ob;
            }
            rome_reply(mp);
```

PUTMBLK requests cause the buffer to be freed when all of the data have been transmitted, so the output buffer is linked back to the internal free queue. In either case (*OUTMBLK* or *PUTMBLK)*, once all the data are transmitted the message is returned to its originator. This code is followed in the source by the *FLUSH* handling described above. Then:

```
            if  (port->writeq.head)
            {
                port->currtxm = port->writeq.head;
                port->currtxp = port->currtxm->b_rptr;
            }
        }
    }
```

the variables are then initialised for the next message at the head of the output queue.

```
    if  (port->currtxm == NULL && todo == '\0')
    {
        WR(IER, IER_RxIE | IER_RLS);
    }
```

the ‘transmit available’ interrupt is disabled only when there are no more message, and no *todo* characters to be sent.

6.5.11 The RETMBLK Message

Queue Handler

```
    rome_auto_reply(mptr);
```

Since the UART uses the immediate data area for *GETMBLK* requests, there is no processing needed to return this buffer.

6.5.12 The main routine

There is no processing in the main routine that has not been covered above in the message descriptions. However, it does introduce another useful ROME routine. In many cases, main routines for processes perform identical logical operations, looping forever receiving messages, decoding the operation code and calling a processing routine to handle that code. This is made easier by the *rome_generic_handler* library routine, which does the decoding automatically, given a table of operations codes:

```
static ROME_HANDLERS serial_routines[] =
{
    {ROME_M_CLOSE, serial_m_close},
    {ROME_M_FLUSH, serial_m_flush},
    {ROME_M_COMMAND, serial_m_cmd}
};

void serial_process(void)
{
    while (1 == 1)
    {
        rome_generic_handler(rome_await_message(0, 0), serial_routines,
                             sizeof(serial_routines) / sizeof(ROME_HANDLERS));
    }
}
```

The other message accepted by the serial module, *COMMAND*, will be described in the next tutorial.

6.6 The Debugging Environment

Debugging code in an embedded system is not easy, particularly when the operating system does not protect individual process's memory and when the system is handling multimedia data.

6.6.1 Philosophy

ROME processes, which enjoy all the benefits of shared memory access for fast data transport, must face the consequences when a process can corrupt memory which can go undetected for many machine cycles. The traditional approach to detecting such problems is to run a process 'under the debugger' and trace instructions and operations until the problem is found. The other traditional approach is to dump the process' state in a *core* file and attempt to reconstruct events leading up to the point of failure.

Neither approach works well for ROME. Breakpointing code only works for non-time-critical applications, where pausing operations for tens of seconds while a user presses keys on a keyboard is an acceptable possibility. For a system handling real-time video, even 30ms delay would disrupt the dataflow to the extent that any 'normal' operation could not be resumed. Likewise, in systems with no external disk storage, dumping core requires a functioning network link, and a protocol stack, which may be the very component that is being corrupted. 'Fixing up' the code to allow the network dump to complete may hide the problem even more.

ROME does not provide a traditional breakpointing debugger. The debugging environment is instead constructed from four components:

(a) a machine-level debugger allows access from the UART to the state of the machine; its registers, processes and memory. This routine tries, in general, to use code that is separate from the main process and runtime code (for example it uses *rome_kprintf* and the polled I/O routines for all its interactions with the user). This routine is tied to various error conditions detected in the ROME core (unhandled interrupts, address or data exceptions).

(b) any routine may, at any time, place a 'trace' record in the system trace table. This records the last 256 events in the system. It may be used, for example, to record the values of variables on entry and exit from an ISR or to monitor the context-switching flow through the system.

(c) the debugger provided in (a) to catch system errors is available as a shared-library interface through the *rome_debug* call. This effectively gives a form of breakpointing, under program control.

(d) through the ROME image symbol table, the debugger can call user-supplied routines for specific environments. For example a module could supply a routine which formats its data structures for human consumption. This simplifies the debugger, which knows only about the core structures and allows a flexible and distributed approach to fault management.

The overall approach is to have the debugger catch errors quickly, and do as little as possible to change the state once they have been detected. The 'shared library' approach to entering the debugger allows a module developer to use the full range of 'C' constructs in making consistency checks on data.

6.6.2 Enabling a debug environment

This section introduces the ROME debugger using the direct entry from the UART interrupt handler. Using the *Perf* project from tutorial III, enable the *SERIAL_UART16550_ENTER_DEBUG* and *ROME_TRACE_INTERRUPTS* options in the project definition, then re-build.

The effect of *ROME_TRACE INTERRUPTS* on the serial interrupt handler is shown above, it compiles in the calls to *rome_add_trace.* The other code was omitted from the earlier description, but is given here. It is part of the inputr handling routine of the serial interrupt service routine:

```
#ifdef SERIAL_UART16550_ENTER_DEBUG
    if (uix == 0 && data == SERIAL_UART16550_ENTER_DEBUG)
    {
        rome_debug();
    }
    else
#endif
```

After rebuilding, do a *make clean* to remove the old object files, then make, install and boot the system. It should produce almost the same output as before (the 'built' string will be different of course) and cache-line effects may change the actual performance values.

```
ROME Initialising.
Copyright 1997 NEC USA Inc.
Built by leslie on pc-rome at Thu Apr  5 13:18:01 2001
Starting the Scheduler.....
perf: Queue-Handling 0.31
```

Pressing the '!' key on the terminal keyboard will now enter the system debugger.

```
!
Entering debugger with SP 0x00ffd7cc FP 0x00ffd7e4
rome debug>
```

At this point the whole system has stopped except for a routine polling the keyboard for input. No interrupts are being processed and no messages are being handled. It is possible to examine and modify memory in the system and to look at the state of the processes.

6.6.3 Debug commands

The set of commands available in the debugger will depend on the architecture of the system (for example the MIPS version does not support call traceback, though the Intel versions do). Also information on register and memory use will vary from board to board. All systems provide the *help* command which lists the available commands.

For the symbolic information to be printed, the system must be linked with the *Create a symbol table* option enabled in the 'Misc' section of the project description.

The *lp* command lists the processes in the system:

```
rome debug> lp
Process console located at 0x00fffe70.
Process serial located at 0x00fff4c0.
Process up located at 0x00ffe310.
Process resp located at 0x00ffd960.
Process perf located at 0x00ffcfb0.
Process timer located at 0x00ffc600.
Process idle located at 0x00ffa450.
Process pointer is currently pointing to resp.
```

depending on exactly when the interrupt for the '!' key is handled, the current process may be one of perf, resp, uptime, console or uart. A particular process may be selected with the *cp* command (change process) and the *pinfo* command prints information on that process:

```
rome debug> cp idle
Switching process pointer to process idle.
rome debug> pinfo
Process idle at 0x00ffa450
Stack at 0x00ffa3d0
EDI 0x00000000 ESI 0x00000000 EBP 0x00ffa3fc ESP 0x00ffa3f0
EAX 0x002c6dfd EBX 0x00000000 ECX 0x00000000 EDX 0x002c6dfe
EIP 0x00014558 CS 0x00000008 EFLAGS 0x00000213
State: 1, Priority: 0
No messages on queue.
```

The output shows the processor registers for the process, its state (runnable) and its current scheduling priority (0).

The *ROME_TRACE_INTERRUPTS* Option turned on interrupt tracing for the UART. This writes a line in the trace table for every entry to, and exit from, the UART ISR. The *trace* command displays the (circular) trace table:

```
rome debug> trace
idle: Start interrupt 0x00000000 at 0x000000c2
idle: End interrupt 0x00000000 at 0x00000000
idle: Start interrupt 0x00000000 at 0x000000c2
idle: End interrupt 0x00000000 at 0x00000000
```

this sample shows a small part of a full trace. The output lists the process that was running at the time of the trace entry, the trace type and the two parameters supplied on the trace call. The debugger understands

certain pre-defined trace types, including the start and end interrupt entries. Other trace types may display as a hexadecimal number. User-defined trace numbers should start at 256 (0x100). Using trace records it is possible to observe the flow of control, and by recording the current process, routines in shared libraries can also be debugged.

In the above trace, note that as the 'perf' process is blocked waiting for the *printf* call to complete, the only runnable process is *idle*, until the reply to the message is sent by the UART ISR.

The *symbols* command displays the local symbol table:

```
rome debug> symbols
__start 0x00010002
_rome_start 0x0001406c
__bssstart 0x00022020
__bssend 0x00022310
_cpu_interrupt_uh 0x0001122c
_cpu_interrupt_error 0x00011080
_icu_exception_handlers 0x00022240
_rome_allow_reschedule 0x00021198
_rome_this_ptr 0x00022300
_rome_run_queue 0x00022304
_cpu_scheduler 0x00010572
_cpu_suspend 0x0001056c
_rome_debug 0x0001058c
```

and the defined symbols can be used to examine memory locations, for example to view code:

```
rome debug> di _rome_start
0001406c: 55              U       PUSH   EBP
0001406d: 89e5            ..      MOV    EBP,ESP
0001406f: 83ec14          ...     SUB    ESP,14
00014072: 57              W       PUSH   EDI
00014073: 56              V       PUSH   ESI
00014074: 53              S       PUSH   EBX
00014075: c745fc00000000  .E..... MOV    fc[EBP],00000000
0001407c: e8dbfbffff      .....   CALL   13c5c
00014081: e886490000      ..I..   CALL   18a0c
       etc.
```

or variables:

```
rome debug> dm.w _rome_run_queue
0x00022304: 0x00ffd960 0x00201000 0x00fffe70 0x1e3697cd
```

This shows the head process (0xffd960, resp) linked into the queue of runnable proceses.

The *message* command will format an area of memory as a ROME message, treating the data area as the *mblk* form. This covers the majority of messages in the system. For other message formats, modules which define the message may provide routines to format their own messages.

Symbols can be used from the debugger to execute such local routines. Using debugger-callable routines allows the debugger to be kept simple, since it does not need knowledge of all the internal data structures of each of the modules, and allows each module control over the display and formatting of its data. More information about the debugger features can be found in the ROME Porting Guide.

7 Tutorial VI: Ethernet ARP

This tutorial presents module designed to sit within a dataflow (unlike the UART module which terminates a flow). The module shows how the ROME message fields are manipulated for zero-copy data flows with protocol processing.

7.1 Functionality

The ETHER_ARP module implements the Internet Protocol layer 3 address to ethernet MAC address translation operation. It also handles the Address Resolution Protocol (ARP). The process operates between one or more 'downstream' ethernet driver processes and one or more 'upstream' network protocol processes (usually only IP). The process is responsible for supplying the ethernet header addresses for outbound packets, and routing them to the correct downstream device. It examines the header for incoming packets and passes them upstream to the correct network layer (or handles them internally).

As with the UART module, ETHER_ARP receives requests from its upstream clients. Unlike UART, most of these requests must be passed down to the appropriate driver. The replies to these requests also pass through the ETHER_ARP process. This is an important aspect of the dataflow model.

7.2 FILEs

Both the *hello* process and the *echo* process used 'C' runtime I/O routines which take a *FILE* * parameter, either explicitly (as with *fgets*) or implicitly (as with *printf*). The *FILE* is an important abstraction for simplifying the downstream flow of data. Apart from the support for character-mode I/O (which is not recommended, but is explained at the end of the tutorial), a *FILE* just contains the linking information to pass messages downstream:

```
typedef struct _file
{
    ROME_PROCESS *dest_pid;        where this connects
    void         *opaque;          returned by open
    int           flags;           eof etc.
    int           fileno;          file number
    int           buffer;          for ungetc
}FILE;
```

The *dest_pid* variable is the result from calling *rome_find_queue* on the supplied filename, and the *opaque* variable holds the *dest_context* field to be inserted into all messages send from this file.

Even when the 'C' standard I/O library is not being used, the FILE is still a useful representation for inter-process flows. The usual interface to the six dataflow messages is through a set of library routines, five of which: *rome_fetmblk*, *rome_getmblk*, *rome_newmblk*, *rome_outmblk*, and *rome_putmblk* all take a pointer to a *FILE* structure as their first argument. The other routine, *rome_retmblk* takes just a message pointer.

7.3 URLs

Within a layered protocol stack, and particularly for IP, there is a fan-out of processes towards the applications. IP supports multiple transport protocols (notably TCP and UDP) and TCP supports multiple user

connections. The connections are distinguished by small integers, representing procotol types and ports. For example each IP packet contains a protocol number which distinguishes TCP traffic from UDP (and ICMP and other IP-based protocols). Similary, ethernet frames are distinguished by a 16bit type field in the header. The ARP process itself does not need any knowledge of the details of most of the protocols, but it does need to pass the data to the appropriate process. The same argument holds for IP, UDP and TCP. When a process opens a dataflow to the ETHER_ARP process it must indicate which protocol packets it wishes to process. This information is passed to the *fopen* routine in the open string parameter, encoded in a form very similar to the 'Uniform Resource Locator' which is the *de facto* standard for representation of resources in the World-Wide Web.

ROME uses the *scheme* part of the URL (the string before the first ':') to identify a process within the image. The terminating ':' is optional if no further information is passed. For example:

```
fp = fopen("console", "rw");
```

contains a minimal ROME URL string consisting of just a process name. The standard schemes defined for the URL do not have any place for information local to the originating system. ROME places this information after the scheme terminator and before the internet syntax part. This is used to hold local port or protocol numbers. The ROME URL syntax is:

```
process:localport//user:password@host:port/url-path
```

where all fields are optional except for the *process*. Standard URLs use an encoding scheme with '%' as the encoding escape character to allow, for example, '@' in a password. The string form is only used on the interface to *fopen* for compatibility with standard C programming conventions. Internally, and in the *OPEN* message, a ROME URL is represented by the following structure:

```
#define ROME_NAMELEN 32
#define ROME_HOSTLEN 64

typedef struct
{
    char  scheme[ROME_NAMELEN];          process name
    uint  port;                          local protocol id
    char  user[ROME_NAMELEN];            remote user
    char  password[ROME_NAMELEN];        remote password
    char  host[ROME_HOSTLEN];            remote hostname
    uint  ipaddr;                        remote host address
    uint  ip_port;                       remote port
    char *urlpath;                       rest of path
}ROME_URL;
```

the values in these fields are not URL-encoded. The *rome_parse_url* routine converts the string form into the structure.

7.4 Upstream, Downstream and Queues

So far, ROME messages have been shown passing between only two processes. The sender had the *src_context* field available for its use and the receiver had the *dest_context* field. When there is a third

process intervening, in theory it has no place to store context information. Any solution that reserves one field per potential process per message does not scale.

If a process on a flow does not require local state it can pass the contexts between its adjacent pairs of processes. However many processes, including ETHER_ARP, need to access context information, and the only option is to place it in the message.

ROME adopts a model that has similar behaviour to the STREAMS module stack, but with a different internal implementation. The link between three processes is represented by a **queue_t** data type:

```
typedef struct
{
    ROME_PROCESS *src;                  uplink pointer
    ROME_PROCESS *me;                   my pid
    FILE         *dest;                 downlink pointer
    ROME_MQUEUE  reads;                 reads list
    ptr          q_ptr;                 local context
}queue_t;
```

this links a upstream process to a downstream file, and contains a pointer to the local context information for the flow.

As each message passes through the process, the source context field is saved (downstream) and restored (upstream) using a local block:

```
typedef struct _context
{
    struct _context *link;              free queue linking
    queue_t         *local_context;     this process' information
    ptr             src_context;        saved source context
}CONTEXT;
```

A pointer to this data structure is placed in the *src_context* field in the downstream direction. These operations are implemented by a pair of routines which perform approximately the same function as the *putq* family of routines for STREAMS services. The *rome_pass_downstream* routine passes a message to the process represented by the *dest* file in a queue structure, saving the necessary context. The *rome_pass_upstream* routine restores the upstream context and passes the reply to the *src* process.

The *CONTEXT* memory is managed from a shared pool across all processes and does not require a *malloc*/*free* overhead once the system is in a steady state.

One major difference between ROME and traditional STREAMS is that ROME does not require a separate scheduler to run STREAMS modules. The core scheduler runs the processes depending on the state of their queues. There is no concept of enabling or disabling queues, though the STREAMS emulation library supports the various routines; they just have no effect.

7.5 Configuration Messages

The ETHER_ARP module is completely portable between all (current) ROME platforms. It has no specific knowledge of the lower level interfaces or drivers. For the most part it operates only on the nine basic dataflow messages. It does, though, need some machine-specific information in order to operate correctly.

Specifically, it needs to know what are the downstream interfaces, and what are their local interface addresses. This is very similar to the *ifconfig* operation on UNIX systems. Rather than define a new set of messages to carry this information, ROME uses a single, simple message, called *COMMAND*. This message carries a text string into the process, where it can be interpreted as required.

The advantage of using a message instead of a shared-library call is that the message can be compiled in to the system even if the module is not present. Because processes are also identified by strings, the *rome_find_queue* call will return an error if the process is absent, and the message will not be sent. If this was a shared library, it would be necessary to conditionally compile in the configuration calls only if the module is present.

The *COMMAND* message works well for setting information, however it does not return any information. In order to extract information from a process, an explicit message must be sent. The network stack defines its own MessageSet, *NetInfo*, which is used between the network modules to pass information around. For example the *NETADDRESS* message is used to read the local MAC (layer-2) address of a driver.

7.6 Finite State Machines

In addition to passing ethernet packets up to IP, the ETHER_ARP module implements ARP – in order to find the ethernet addresses in the first place. If an IP-Address to MAC address mapping is not known, the process sends out an ARP request to the network and waits for the reply. Since packets may get lost in the network, the process must also re-transmit packets on timeouts. The replies are cached to reduce the number of requests sent to the network. This means that a given cache entry can be in one of a number of possible ‘states’ (such as waiting for a reply to an ARP request). Transitions between states are (usually) triggered by messages arriving for the process, for example a timeout to cause a re-transmission.

The natural representation of this mechanism is as a formal Finite State Machine. In the same way that ROME support message dispatching, through the *rome_generic_handler* routine, there is basic support for FSMs. However, this will not be covered explicity in this tutorial, to keep the focus on the message-passing interface.

7.7 Events

Some processes can generate asynchronous ‘events’. A typical event of this form is the ejection or insertion of removable media. As well as the obvious floppy disks, this could also include PCMCIA-based interface cards, including ethernet cards. In a similar apprach to configuration commands, such events are passed around the system using messages. All events are represented by a single message with opcode *EVENT*. The first parameter of each event message is a ‘type code’ which identifies the exact event being supplied.

7.8 Source Code

As with the UART tutorial, the code will be described message by message. There are two differences from UART: the module does not contain an interrupt handler; and the module must process both messages (from upstream) and replies (from downstream).

7.8.1 Data Structures

Each flow through the ETHER_ARP process is represented by an *queue* type (see above); each configured interface below ETHER_ARP is represented by a *ARP_DOWNINT* type and each ARP entry by a *ARPENTRY* type.

```
typedef struct
{
    int          state;              state of entry
    mblk_t      *mp;                 mblk of pending ARPs
    int          nr_arp;             number of ARPs to try
    int          epoch;              set time
    int          ifno;               interface to send it on
    uint         ip;                 the IP address
    MACADDR      mac;                MAC address
}ARPENTRY;

typedef struct
{
    FILE        *arp_fp;             file descriptor for ARPs
    char         ifname[16];         interface name
    int          ifno;               interface number
    MACADDR      mac;                our MAC address
    uint         subnet_address;
    uint         ip;                 our IP address
    uint         ipmask;             and mask
    ARPENTRY     broadcast;          for broadcast pkts
    ARPENTRY    *local;              The ARP table
    int          localsize;          size of the table
}ARP_DOWNINT;
```

the array of interfaces is held in local static storage:

```
static ARP_DOWNINT *di[EARP_MAX_INTERFACES];
```

The *di* entries are assigned as the configuration command messages are processed.

7.8.2 The Command Message

The ETHER_ARP process may be connected to multiple ethernet interface cards in a system. These may be different types of cards, with different interface drivers, or multiple instances of the same card sharing the same driver. Command messages are used to configure the interfaces for a particular machine. They are all handled by the main process.

interface: interface interface-index device-url interface-number

```
if (sscanf(inc, "interface %d %s %d", &i, ifname, &i0) == 3)
{
    ROME_MESSAGE *m;
    int j;
    ROME_URL dest;
```

The interface message links an interface index number used by IP to a particular ethernet card in the system. The card is identified by a device URL and a 'selector' (the interface number) within that device. For example a machine with two cards sharing the same driver might be numbered '0' and '1' on the same URL.

```
di[i] = rome_alloc(sizeof(ARP_DOWNINT), 'e', 1);
strcpy(di[i]->ifname, ifname);
di[i]->ifno = i0;
strcpy(dest.scheme, ifname);
dest.port = i0;
dest.ip_port = ETHER_TYPE_ARP;
```

In the URL passed to an ethernet interface, the device selector is in the *port* field, and the *ip_port* field is used to contain the protocol selector, in this case *ETHER_TYPE_ARP* (0x0806) to select only the ethernet ARP packets.

```
if ((di[i]->arp_fp = rome_open_url(&dest, "rw")) == (FILE *)NULL)
{
    rome_fatal("Couldn't open ethernet interface\n");
}

if (i+1 > dc) dc = i+1;
```

By using the *FILE* interface to the device, the ETHER_ARP process allows the ethernet driver to set its own context variables for the flow and return them here. *rome_open_url* is the 'internal' routine used to open a file using a pre-parsed (or explicitly constructed) URL. The removes the need to convert a URL back to a string form in order to use *fopen* (which would then have to parse it again).

```
    earp_reinit(i);
    for (j = 0; 10 > j; j++)
    {
        m = (ROME_MESSAGE *)rome_alloc(sizeof(ROME_MESSAGE), 0, 1);
        m->src_context = (ptr)i;
        rome_getmblk(di[i]->arp_fp, m);
    }
}
```

Simply opening the interface FILE does not cause packets to arrive at the process. The ETHER_ARP process must explicitly request packets on the file. In this case, up to ten incoming ARP messages will be queued within the system. Note that it is essential that the *rome_getmblk* routine does not wait for the reply, otherwise the process would block after the first request was issued. Also, since these messages will remain in the system after this routine returns, they must be allocated off the heap, not the local stack. In order to determine from which interface a reply has come, the process puts the interface index number into the *source* context field of the message.

The code called above to (re-)initialise the interface is:

```
static void earp_reinit(int i)
{
    ROME_MESSAGE m;
    ROME_T_NETADDRESS *na = RCAST(ROME_T_NETADDRESS, m);
```

```
        m.opcode = ROME_M_NETADDRESS;
        m.dest_context = di[i]->arp_fp->opaque;
        rome_sendwait(&m, di[i]->arp_fp->dest_pid);
```

There is no shared-library interface for the *NETADDRESS* message (since it is only used in one place), so the message is constructed explicitly. In order to identify the interface to the lower layer, the correct destination context must be placed in the message. This is copied from the *FILE* structure, where it was placed by *rome_open_url*. *rome_sendwait* is a library routine that completes the fields of the message, sends it to the destination, and waits for the reply. In this case, therefore, the message can be put on the local stack.

```
        if (m.m_errno)
        {
            rome_kprintf("EARP: no MAC address for interface %d\n", i);
            rome_fatal("interface error");
        }
        memcpy(di[i]->mac, na->mac, 6);
```

In theory, no ethernet interface can run without knowing its own MAC address. If, for some reason, the destination process returns an error, then the system halts through the *rome_fatal* routine. This is an example of the debugging philosophy described in the previous tutorial. Since this probably represents a typing error in the configuration command, rather than try and hide the error, the system stops as quickly as possible to allow a user to examine the state and correct the problem.

```
        if (na->events_generated)
        {
            int j;
            ROME_MESSAGE *tm;

            for (j = 0; 2 > j; j++)
            {
                tm = (ROME_MESSAGE *)rome_alloc(sizeof(ROME_MESSAGE), 0, 1);
                tm->src_context = (ptr)i;
                rome_get_event(di[i]->arp_fp, tm, rome_this_ptr->prio);
            }
        }
    }
```

If the interface reports that it supports events (see above), then the ETHER_ARP process sets up a queue of event requests to that process, in a similar manner as the data requests.

config: config interface-number ipaddress netmask

```
    if (sscanf(inc, "config %d %I %I", &i, &i0, &m0) == 3)
    {
```

The config command completes the initialisation of an interface, by supplying the local IP address and netmask for that interface.

```
    if (i < 0 || i > EARP_MAX_INTERFACES || di[i] == NULL)
    {
        rome_fatal("EARP: Bad interface in config");
    }
```

The interface must previously have been set up by an *interface* command.

```
    di[i]->ip = i0;
    di[i]->ipmask = m0;
    di[i]->subnet_address = i0 & m0;
    di[i]->broadcast.state = ARP_ENTRY_OK;
    di[i]->broadcast.ifno = i;
    memset(di[i]->broadcast.mac, 0xff, 6);
    di[i]->localsize = ntohl(~m0) + 1;
    di[i]->local = rome_alloc(di[i]->localsize*sizeof(ARPENTRY), 'e', 1);
}
```

The netmask is used to determine the size of the routing subnet attached to that interface, and so to calculate the possible size of the ARP table.

7.8.3 The EVENT Message

Queue Handler

```
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

Since the ETHER_ARP process does not itself generate events, any event requests are passed downstream to the driver.

7.8.4 The EVENT Reply

Queue Handler

```
if ((uint)mptr->src_context < EARP_MAX_INTERFACES)
{
    return ROME_NOT_HANDLED;
}
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

The ETHER_ARP process makes use of the fact that *EVENT* messages passed downstream will have a pointer value in the *src_context* field, which cannot be a small integer. Therefore if the value is one of the interface index numbers, it must have originated within the ETHER_ARP process itself, in which case it should be handled in the main process.

Main Process

```
ROME_T_EVENT *event = RCASTP(ROME_T_EVENT, mptr);
int ifno = (int)mptr->src_context;

#ifdef ROME_E_PCMCIA_CARD_INSERT
if (event->event_type == ROME_E_PCMCIA_CARD_INSERT)
{
    earp_reinit(ifno);
}
#endif
rome_get_event(di[ifno]->arp_fp, mptr, rome_this_ptr->prio);
```

This code re-initialises the MAC address for a newly-inserted PCMCIA ethernet card.

7.8.5 The OPEN Message

Main Process

```
ROME_T_OPEN *o_msg = RCASTP(ROME_T_OPEN, mptr);
queue_t *q = (queue_t *)rome_alloc(sizeof(queue_t), 0, 1);
ROME_URL dest;
int ix = o_msg->openurl->port;

if (ix >= dc || di[ix] == NULL)
{
    rome_free(q);
    mptr->m_errno = ENXIO;
}
```

The URL contains the interface index in the *port* field and the protocol selector in the *ip_port* field. The index must correspond to a previously-configured downlink.

```
else
{
    strcpy(dest.scheme, di[ix]->ifname);
    dest.port = di[ix]->ifno;
    dest.ip_port = o_msg->openurl->ip_port;
    q->src = mptr->src;
    q->me = rome_this_ptr;
    q->dest = rome_open_url(&dest, "rw");
    mptr->dest_context = q;
}
rome_reply(mptr);
```

If it does, the index is used to construct a new URL based on the downward interface name. This is opened as a file which is used to initialise the three-way queue structure, which is returned as the local context to the caller.

7.8.6 The CLOSE Message

Main Process

```
queue_t *q = (queue_t *)mptr->dest_context;

fclose(q->dest);
rome_free(q);
rome_reply(mptr);
```

The processing for the *CLOSE* message is simpler than for the UART code, as the process does not keep a list of read requests. It is assumed that the higher layer process does not try to close a file that has outstanding write requests. If so, the code would need to check that no messages for this queue are currently held awaiting ARP replies. As it stands, the code simply closes the downstream file (which may cause it to return message upwards) and frees the queue structure.

7.8.7 The FLUSH Message

Queue Handler

```
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

It is known that the IP code does not generate *FLUSH* messages, and as that is the only process that can cause data to be buffered inside the ETHER_ARP process, the message can be simply passed down to the lower layer.

7.8.8 The FLUSH Reply

Queue Handler

```
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

Since all *FLUSH* messages were passed downstream, all replies must be passed upstream. The ETHER_ARP code does not generate *FLUSH* requests itself, so there is no need to check for replies to local messages.

7.8.9 The FETMBLK and GETMBLK Messages

Queue Handler

```
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

Requests for filled buffers are passed downstream, since the ethernet driver has sufficient information (the ethertype protocol selector) to assign incoming frames to individual flows.

7.8.10 The FETMBLK and GETMBLK Replies

Queue Handler

```
if ((uint)mptr->src_context < EARP_MAX_INTERFACES)
{
    return ROME_NOT_HANDLED;
}
```

Replies to messages that were sent from within the ETHER_ARP process itself are passed to the main process (below).

```
mptr->b_rptr += sizeof(ETHER_HEADER);
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

The ethernet driver returns the 'read pointer' for the incoming data at the start of the ethernet frame. The ETHER_ARP process modified this pointer to point to the start of the layer-3 protocol header before passing the message upwards. In this way the IP code is unaware of the size of any header on the message (so the IP code is independent of the layer-2 addressing modes).

Main Process

```
ETHERARP *arph = (ETHERARP *)mptr->b_rptr;
int ifno = (int)mptr->src_context;

if (ntohs(arph->ar_op) == ARPOP_REPLY)
{
    ARPENTRY *entry = earp_arp_get(*(uint *)arph->arp_spa);

    if (entry)
    {
        (*earp_p_machine[entry->state])(ARPEVENT_RESOLVED, entry, mptr);
    }
}
```

An incoming message from the net could be a reply to a request sent out from this machine. If so, the Finite State Machine for that request is run with the *ARPEVENT_RESOLVED* event.

```
else if (ntohs(arph->ar_op) == ARPOP_REQUEST)
{
    uint forme = *(uint *)arph->arp_tpa;

    if (forme == di[ifno]->ip)
    {
```

or, it could be a request from another machine for one of our interface addresses. The code only replies if the request came in on the same interface as its configured IP address, to prevent MAC addresses crossing subnet or VLAN boundaries.

```
ROME_MESSAGE reply_msg;
ETHERARP  *arp_reply;

reply_msg.src_context = (ptr)ifno;
rome_newmblk(di[ifno]->arp_fp, &reply_msg, sizeof(ETHERARP));
rome_await_message(&reply_msg, 0);
```

In order to send a response over the network with the MAC address, the process must request a buffer from the ethernet driver. It sends a *NEWMBLK* message downstream using its *FILE* pointer, requesting a buffer large enough to hold an *ETHERARP* packet. The source context is marked with the interface number (so the reply is passed back by the Queue Handler). The process then waits for the reply.

```
arp_reply = (ETHERARP *)reply_msg.b_rptr;
arp_reply->ar_hrd = arph->ar_hrd;
arp_reply->ar_pro = arph->ar_pro;
arp_reply->ar_hln = arph->ar_hln;
arp_reply->ar_pln = arph->ar_pln;
arp_reply->ar_op = htons(ARPOP_REPLY);
memcpy(arp_reply->arp_sha, di[ifno]->mac, 6);
memcpy(arp_reply->arp_tha, arph->arp_sha, 6);
*(uint *)arp_reply->arp_spa = *(uint *)arph->arp_tpa;
*(uint *)arp_reply->arp_tpa = *(uint *)arph->arp_spa;
```

the reply buffer, starting at *b_rptr* (which is the same as *b_wptr* for an empty buffer), is cast to the appropriate type and the fields are filled in from the original request and the local information.

```
reply_msg.b_wptr += sizeof(ETHERARP);
rome_putmblk(di[ifno]->arp_fp, &reply_msg);
rome_await_message(&reply_msg, 0);
```

the *b_wptr* field is incremented to point to the end of the valid data, and the message is sent using the *rome_putmblk* routine. In order to simplify the processing of ARP responses (and to allow the message to be put on the local stack) the process waits for the reply. Since this uses *PUTMBLK*, the ethernet layer will automatically free the buffer after it has been transmitted.

```
    }
}
rome_retmblk(mptr);
rome_getmblk(di[ifno]->arp_fp, mptr);
```

However, the buffer which contains the original request must be explicitly returned to the driver, since it was received with a *GETMBLK* message. Once the buffer has been released, the same message can be re-used to request another frame.

7.8.11 The NEWMBLK Message

Queue Handler

```
mptr->b_wptr += sizeof(ETHER_HEADER);
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

Requests for empty buffers are passed to the driver. The requested length is increased to allow room for the (fixed-size) ethernet header to be added later.

7.8.12 The NEWMBLK Reply

Queue Handler

```
if ((uint)mptr->src_context < EARP_MAX_INTERFACES)
{
     return ROME_NOT_HANDLED;
}
mptr->b_wptr = mptr->b_rptr += sizeof(ETHER_HEADER);
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

when a reply to a *NEWMBLK* message is received, there are two possibilities. If the block was requested by ETHER_ARP itself (for ARP responses), the message is handled by the main process. The local requests are detected by having the *src_context* field contain a small integer rather than a pointer to a queue structure. In this case, the *ROME_NOT_HANDLED* indication is returned to the core, to cause the message to be scheduled for the main process.

If the message was a request by a network process above ETHER_ARP, the pointers are moved within the buffer to leave room for the ethernet header, then the reply is then passed upstream.

7.8.13 The OUTMBLK and PUTMBLK Messages

Queue Handler

```
queue_t *q = (queue_t *)mptr->dest_context;

if (mptr->b_rptr != mptr->b_wptr)
{
```

the handler only looks at messages that have any actual content. This automatically bypasses the *PUTMBLK* messages used to return unused buffers to the driver.

```
    mptr->b_rptr -= sizeof(ETHER_HEADER);

    if (M_IPDATA == mptr->b_type)
    {
        ETHER_HEADER *eh = (ETHER_HEADER *)mptr->b_rptr;
        ARPENTRY *entry = earp_arp_get(mptr->b_immed);
```

the IP layer is responsible for IP routing, and determining the correct interface for a message. This gives two possible cases for a message: either the address in the IP header is to be used as the destination address (i.e. inside the subnet); or the packet is destined for a gateway on the subnet, and the IP address in the header should *not* be used to route the packet. IP solves this issue by using the *b_immed* field in the message to contain the IP (version 4!) address of the next-hop. To signal this to the ETHER_ARP layer, the message type is set to *M_IPDATA*. The ETHER_ARP queue handler uses this value to find the destination MAC address.

```
if (entry == NULL)
{
    rome_kprintf("ARP: %I not in our subnet?\n", mptr->b_immed);
    rome_fatal("ARP Routing Error");
}
```

because of this two-level scheme, the address should always lie within the subnet, it is an error in the IP layer if it does not.

```
if (ARPSTATE_OK != entry->state)
{
    return ROME_NOT_HANDLED;
}
memcpy(eh->ether_dhost, entry->mac, 6);
memcpy(eh->ether_shost, di[entry->ifno]->mac, 6);
```

if the MAC address of that IP address is not known, the message is passed on the main process. Otherwise the addresses are set in the ethernet header (which is made accessible by moving the *b_rptr* field backwards in the buffer).

```
    }
}
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

once the addresses are set, the message is passed on to the driver.

Main Process

```
ARPENTRY *entry = earp_arp_get(mptr->b_immed);

(*earp_p_machine[entry->state])(ARPEVENT_LOOKUP, entry, mptr);
```

The main process is scheduled only when the MAC address is of the destination is not known. The code injects a *LOOKUP* event into the state machine for that IP address.

7.8.14 The OUTMBLK and PUTMBLK Replies

Queue Handler

```
if ((uint)mptr->src_context < EARP_MAX_INTERFACES)
{
    return ROME_NOT_HANDLED;
}
mptr->b_wptr = mptr->b_rptr += sizeof(ETHER_HEADER);
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

Here too, replies to *OUTMBLK* and *PUTMBLK* messages could have come from locally-generated traffic identified by the *src_context* field, or they could have come from an upstream process.

Main Process

There is no explicit handler for these replies in the main process. The messages cause the process to resume execution from the *rome_await_message* call in the *F/GETMBLK* reply code above.

7.8.15 The RETMBLK Message

Queue Handler

```
rome_pass_downstream(q, mptr);
return ROME_HANDLED;
```

returned buffers are passed down to the driver which originated them.

7.8.16 The RETMBLK Reply

Queue Handler

```
if ((uint)mptr->src_context < EARP_MAX_INTERFACES)
{
    return ROME_NOT_HANDLED;
}
mptr->b_wptr = mptr->b_rptr += sizeof(ETHER_HEADER);
rome_pass_upstream(mptr);
return ROME_HANDLED;
```

Here too, replies to *RETMBLK* messages could have come from locally-generated traffic identified by the *src_context* field, or they could have come from an upstream process.

Main Process

There is no explicit handler for these replies in the main process. The messages cause the process to resume execution from the *rome_retmblk* call in the *F/GETMBLK* reply code above.

7.8.17 The TIMEOUT Reply

The ARP Finite State Machine (which has not been described here) uses timeout calls to control request retransmissions. The timeout replies are handled by a line in the handlers list:

```
{ROME_R_TIMEOUT, timer_tmhandler}
```

which automatically calls the *timer_tmhandler* routine, which should be familiar from the previous material.

7.8.18 The main process

The main process is constructed around a call to *rome_generic_handler* in the same way as the serial module (but with a rather longer list of messages).

8 Tutorial VII: Network Echo

This tutorial. unlike the previous ones, requires specific PC hardware and may not work in your environment.

This final tutorial brings the networking components together by re-working the 'echo' module to process data from the network. The focus for this tutorial is using the module pages to construct a system from the documented components. The aim is to attach the IP stack under the console process and use the 'telnet' protocol to connect to the machine over the ethernet.

The process structure is shown in figure 4. How these modules are known to be required is described in the next section.

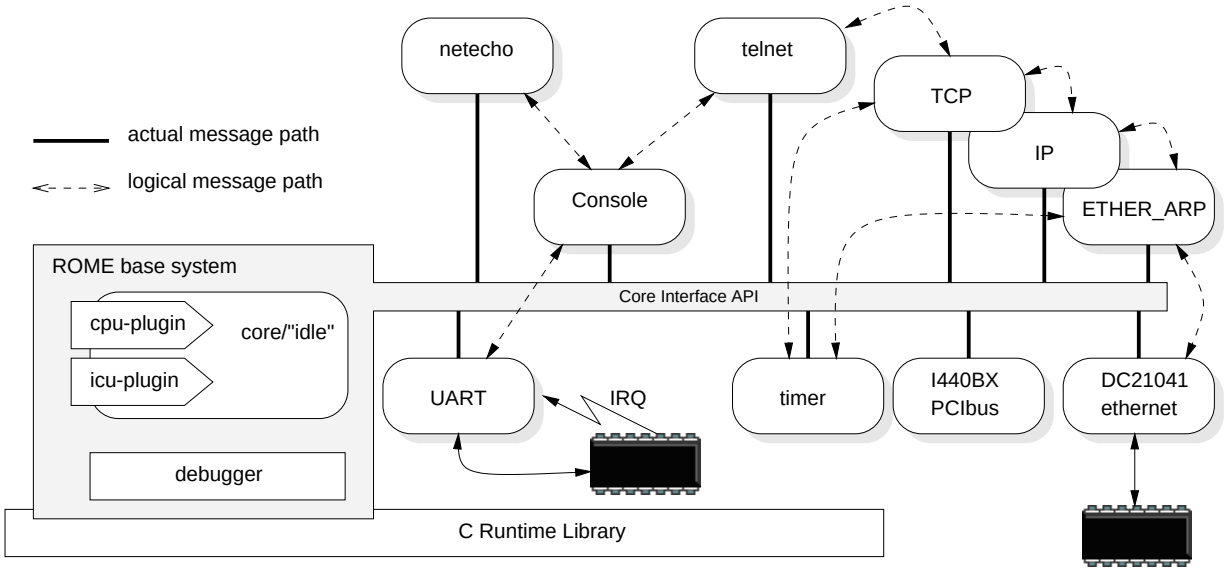

Figure 4: Module Structure for Tutorial VII

8.1 Choosing the Modules

Building a complete system for an embedded application generally requires a detailed knowledge of the platform on which the application is going to run. The previous tutorials were carefully chosen to run on "almost all" PC-based systems. However, as soon as applications require specific hardware, the picture changes. For the PC development of the ROME system, the hardware platform was a NEC Direction SPB-450 PC. This uses an Intel 'Seattle' motherboard with a 450MHz Pentium-II processor. The motherboard provides four PCIbus slots under the control of the Intel I440BX chipset. The PC does not come with onboard ethernet, and a Kingston KNE110TX PCIbus card was used to make a local ethernet connection.

All the ROME modules describe how they are to be used and what they can be used for. The *ether_dc21041_liteon* module describes itself as a driver for the LiteOn clone of the DC21041 'Tulip' chip, as used in the Kingston KNE110TX ethernet interface card. As the card is a PCI card, a PCI bus controller module must also be included, in this case the *pci_i440bx* module.

At the other end, the *telnet* module has a requirement on the *TCP* module, which itself uses the *IP* module. Finally, the *ETHER_ARP* module is the link between IP and an ethernet driver. The ethernet and ETHER_ARP modules both specify the *NetInfo* message set as a requirement (as described above).

This gives enough information to generate the project description in RTB. Using the *echo* project of tutorial II, add the following into the project:

MessageSet: NetInfo, Timer

Modules: Driver.PCI.pci_i440bx, Driver.Timer.timer_pc, Driver.Ethernet.ether_dc21041_liteon, Network.Protocol.ETHER_ARP, Network.Protocol.IP, Network.Protocol.TCP, telnet

As before, the *echo* process is tied in to the console through the standard C I/O files, by marking the process to undergo C runtime initialisation.

8.2 Initialising IP

The main application file, *echo.c*, needs a small modification to complete the network initialisation. The previous chapter described the *COMMAND* message used to link the IP protocol stack to a particular interface and interface address. This message must be sent to IP to complete the link. The interface address is the "earp" process; the IP address is statically linked into the process in this (simple) example. You should choose an IP address allocated to your domain. If it is not in the same subnet as the machine which will start the telnet connection, you will also need to give a gateway address.

8.3 Source Code

Since you created the *echo* module, it is already writable within the module. Remember to edit the version in the *hello/Modules/Echo* direcroty, not the *Sources* directory. For example, to set the machine's IP address to 138.15.103.243, add the lines:

```
rome_send_command("earp", "interface 0 ether 0");
rome_send_command("ip",
    "config 0 138.15.103.243 255.255.255.0 earp:0 2048");
```

to the main echo process. The format of the first string is explained in the ETHER_ARP tutorial, and the format of the second string is given in the *IP* module documentation. The *rome_send_command* routine formats and sends a *COMMAND* message to the specified process.

8.4 Build, Load and Run

The program is built and loaded in the usual way. From another machine on the same network (for example the development host) a TCP connection can be opened over the ethernet:

```
telnet 138.15.103.243
Trying 138.15.103.243...
Connected to 138.15.103.246.
Escape character is '^]'.
logon: rome
echo foo
echo: foo
echo this is a test of network echo
```

```
>echo: this is a test of network echo
^]
telnet> close
Connection closed.
```

The network link over telnet prompts for a login name, to prevent accidental connections to development systems. Only the string ‘rome’ is accepted. The console automatically redirects all standard output to the network link. The connection is closed by entering control-] and the ‘close’ command to the (local) telnet process.

9 Legacy Applications and the C Runtime

In an ideal world the previous tutorial would have completed all the functionality for the ROME system. There remain, though, a few cases which do not fit into this model. There are also some applications which might seem not to fit, but do indeed work well with ROME dataflows.

9.1 Traditional Buffering

The main problem comes with legacy applications which use the C runtime library routine to read data. An example is the *echo* program above. It uses *fgets* to read data into a buffer it supplies. As expected, the C library passes this to the destination process as a FETMBLK message:

```
char *fgets(char *s, int n, FILE *stream)
{
    ROME_MESSAGE message;
    char *rv = s;

    if  (stream->buffer != EOF)
    {
        *s++ = stream->buffer;
        stream->buffer = EOF;
        n--;
    }
    mblk_setup(&message, s, n);
    rome_fetmblk(stream, &message);
    rome_await_message(&message, 0);
    return ((message.m_errno == EOF) ? NULL : rv);
}
```

The code for the *stream->buffer* variable handles a single character pushed back onto the stream by *ungetc*. The remainder of the code formats a FETMBLK message, sends it downstream and waits for the reply. The problem comes when this meets a module which operates in the GETMBLK mode, for example IP. Ultimately, there is no choice but to copy the data.

In *echo*, these two dataflows meet in the console module, which is where the data copy occurs. The console module also has to strip off the prefix for line-by-line input.

9.2 Buffered Output

The output direction is less of a problem. There are two cases, *printf*, which uses an internal buffer, and *fputs* which is given a string.

9.2.1 printf

The printf routine works very well with ROME dataflows, since it can use *NEWMBLK* and *PUTMBLK*. The routine which integrates the different interfaces is *vfprintf*. This is called by the other routines, such as *printf* and *fprintf*, with a uniform interface:

```
int vfprintf(FILE *stream, const char *format, va_list args)
{
    ROME_MESSAGE msg;
    int size;

    rome_newmblk(stream, &msg, MAX_PRINT_BUFF);
    rome_await_message(&msg, 0);

    if ((size = gvsprintf(msg.b_wptr,
            (char *)format, args, MAX_PRINT_BUFF)) >= 0)
    {
        msg.b_wptr += size;
        rome_putmblk(stream, &msg);
        rome_await_message(&msg, 0);
    }
    return size;
}
```

the routine blocks twice in *rome_await_message* until the reply is received, once for the *NEWMBLK* and once for the *PUTMBLK*. This style of programming shows the advantage of handling the *NEWMBLK* in the Queue Handler, which saves a pair of context switches on every call to *printf*.

9.2.2 fputs

The *fputs* routine generates a *OUTMBLK* message:

```
int fputs(const char *s, FILE *stream)
{
    ROME_MESSAGE message;
    int len = strlen(s);

    if (NEQBIT(stream->flags, FILE_WRITE_FLAG))
    {
        errno = EACCES;
        return EOF;
    }
    mblk_setup(&message, (uchar *)s, len+1);
    message.b_wptr += len;
    rome_outmblk(stream, &message);
    rome_await_message(&message, 0);
    return 0;
}
```

The routine constructs an *mblk* with just enough space to contain the supplied string (allowing for the terminating *NUL*). This poses a small problem in the console process where the originating process name needs to be added. With *printf*, space has been reserved at the start of the buffer (in exactly the same way

as the IP module reserved its header space). Under these circumstances (and others, for example when multiple lines are passed in the same buffer), the console code will copy the data to another buffer. This merely emphasises that the zero-copy features of ROME are not automatic for all programming styles.

This completes the tutorial material.