ROME

The ROME module contains the machine-independent scheduling core for the ROME system. It is a mandatory module for all systems.

Process Information

Prototype Name	idle
Process Priority	0
Link Order	last

Module Options

ROME_CHECK_POINTERS	Enables pointer checking of the destination process pointer within the *rome_send_message* routine. The address range for valid pointers must be set in the target file before this option is enabled.
ROME_NO_STDIO	Prevents the process initialisation routine from calling *_main* to initialise standard I/O services. This is for use with a minimal C-library that does not support standard I/O services. The system will execute a *rome_fatal* if any of the processes are marked are using stdio.
ROME_DISABLE_KPRINTF	Removes some code for the `kprintf` functions. This reduces code-size of the target file but also eliminates *all* outputs via the `kprintf` call (e.g. debug messages, init messages etc.). This option should be used to build a *final target* only.
ROME_TRACE_CXSWITCH	Causes a ROME_TT_CXSWITCH trace record to be written for every context switch between processes.
ROME_TRACE_INTERRUPTS	Causes a pair of ROME_TT_STARTINT and ROME_TT_ENDINT trace records to be written for each interrupt handled within the system (needs supprt from the individual drivers to implement this).
ROME_TRACE_MEMORY	Causes a ROME_TT_ALLOC or ROME_TT_FREE trace record to be written for each bloack of memory allocated or freed respectively.
ROME_TRACE_MSGS	Causes message-processing operations to generate trace records. Depending on the exact operation, one or more of the following trace

types will appear: ROME_TT_AWAIT{1, 2, 3}; ROME_TT_SEND{1, 2}.

Target File Definitions

ROME_HAS_SYMBOLS	This symbol is defined automatically when the symbol-generation option is set in the project file. If it is not defined, a dummy symbol table is defined within the ROME module.
ROME_MAX_PPTR	The highest machine address in the system that can be a valid address for a process pointer.
ROME_MIN_PPTR	The lowest machine address in the system that can be a valid address for a process pointer.
ROME_RAMSIZE	The size of the available memory (in bytes) for the ROME system.

Data Definitions

rome.h contains type definitions for the following exported data structures:

FILE	The data structure representing a open file structure. The *dest_pid* field is the destination for messages sent using this file, and the *opaque* pointer is set in the *dest_context* field of all messages. The *flags* field contains the open mode and error bits for the file. The *fileno* is the file's index in the process' file table and the *buffer* area is used by *ungetc* for character buffering.
ROME_INIT_PROC	This data structure is shared between the ROME module, the cpu plugin and the init module generated by RTB. If the layout of this data structure is changed, RTB must also be changed to match it.
ROME_MEMORY	The header of an allocated block in memory, immediately preceding the pointer returned to the user. The *type* value is supplied by the caller to identify the memory block in the debugger, while the *who* variable is automatically set to the process which called *rome_alloc* for this memory.
ROME_MESSAGE/mblk_t	The basic inter-process message format and streams dataflow definition. The structure contains three parts: the *link*, *dest*, and *priority* fields are used by the core for queueing and scheduling the message; the *src*, *m_errno*, *opcode*, *dest_context* and *src_context* fields are common to all operations and transmit context information with the message; the use of remaining fields depends on the particular message opcode. For the STREAMS operations the rest of the message looks like a pair of mblk_t and dblk_t structures, having the standard fields for read and write pointers etc. plus two extra 4-byte fields available for short 'immediate' operands Other operations see a 32-byte area which can be cast to any suitable data structure.

ROME_MQUEUE	An opaque handle to a object capable of sending and receiving ROME messages.
ROME_PROCESS	The machine-independent data definition of a process within ROME. The internal format of the process structure is needed only within the ROME module, the cpu-plugin and the parts of the C library handling per-process structures (for example the file table).
ROME_TRACE	The layout of a trace entry. The *address* and *type* fields are determined by the call to *rome_add_trace*. The *current* field holds the process pointer at the time of the trace call, and the *spare* field holds an additional parameter on the trace call.

The *rome_symbols.h* file defines the ROME_SYMBOLS type containing a *name* and its coresponding *address*. The table is used in the debugger and in modules which use symbols (for example language interpreters).

Process Operation

Main Process: *idle_process*

The idle process must be present in every system, to give the scheduler a process to run when all other processes are in wait state. It does not have to perform any computation, though in ROME it increments a counter in a loop to give a measure of the amount of idle time.

Shared Library Macros and Routines

The following routines with *rome_* prefixes are defined in other modules: *rome_add_handler*, *rome_end_critical* and *rome_start_critical* in the ICU_*xxx* module; and *rome_add_trace* and *rome_debug* in the CPU_*xxx* module. See also the rome_if module for API routines also prefixed with *rome_*.

BIT and FIELD

ROME defines the following macros for bitfield support:

BIT(*_num*)	A 32-bit value with a single bit set at position *_num* (0..31).
BITFIELD(*_num*, *_w*)	A 32-bit value with *_w* bits set starting at position *_num*. Note that **BIT**(6) is equivalent to **BITFIELD**(6, 1).
BITM(*_num, _v*)	A 32-bit value with the value *_v* starting a position *_num*. Note that **BIT**(6) is equivalent to **BITM**(6, 1).
CLEARBIT(*_var*, *_bit*)	Sets the bit at mask *_bit* in the variable *_var* to ‘0’.
EQBIT(*_var, _b*it)	Returns TRUE if *_bit* is set (‘1’) in the variable *_var*.
EQFIELD(*_var*, *_bitf, _m*)	Returns TRUE if the field defined by the mask *_m* equals the value *_bitf*.
NEQBIT(*_var, _bit*)	Returns TRUE if *_bit* is unset (‘0’) in the variable *_var*.
NEQFIELD(*_var*, *_bitf*, *_m*)	Returns TRUE if the field defined by the mask *_m* does not equal the value *_bitf*

SETBIT(*_var*, *_bit*)	Sets the bit at mask *_bit* in the variable *_var* to ‘1’.
SETFIELD(*_var*, *_bitf*, *_m*)	Sets the field defined by the mask *_m* in the variable *_var* to the value *_bitf*.

An example of the use of the bitfield macros is to examine values within parts of a 32-bit word. Suppose the bits at position 13..15 in *status_variable* are used to contain status values of: OK (0); soft-error (1); hard-error (5); and fatal (7). Then the bit operations can be used as follows:

```
#define STATUS_MASK BITFIELD(13,3)
#define STATUS_OK BITM(13,0)
#define STATUS_SOFT BITM(13,1)
#define STATUS_HARD BITM(13,5)
#define STATUS_FATAL BITM(13,7)

if (EQFIELD(status_variable, STATUS_MASK, STATUS_OK) {
        . . . .
```

which avoids unreadable shift-and-mask operations in the main code.

RCAST and RCASTP

In order to access the opcode-dependent data area of a ROME message, two data-conversion macros are provided.

```
(_t *)RCAST(
        anytype _t,
        ROME_MESSAGE _m)

(_t *)RCASTP(
        anytype _t,
        ROME_MESSAGE *_m)
```

The first parameter is an arbitrary type, usually one of the **ROME_T_xxx** type defined in a message set. The second parameter is either a message (RCAST) or a pointer to a message (RCASTP). The result is a pointer to the message data area, cast into the correct type. For example, to access the parameters passed in an OPEN message *mptr*, the following code may be used:

```
ROME_T_OPEN *oopen = RCASTP(ROME_T_OPEN, mptr);

if (oopen->mode == . . . ) {
```

It is the caller’s responsibility to ensure that the supplied type is compatible with the message being passed.

rome_alloc

```
char *rome_alloc(
        uint size,
        uint type,
        uint clear)
```

The *rome_alloc* routine allocates an area of memory of at least *size* bytes and returns a pointer to the start. The *type* field is used by the debugger when tracing memory use. The *clear* parameter is a boolean values and determines whether (TRUE) or not (FALSE) the allocated memory is set to zero before returning the pointer to the user.

rome_auto_reply

(void) *rome_auto_reply*(
  ROME_MESSAGE **_m*)

The *rome_auto_reply* macro returns a message to its source from within a queue handler. It sets up the reply fields and executes a **return** statement.

rome_await_message

ROME_MESSAGE **rome_await_message*(
  ROME_MESSAGE **which*,
  int *poll*)

The *rome_await_message* routine controls process scheduling during inter-process communication. The two parameters *which* and *poll* control the operation of the routine as follows:
If *which* is non-NULL and *poll* is FALSE, the process is suspended until the message *which* is placed on its queue. The mesage is removed from the queue and returned as the result.
If *which* is non-NULL and *poll* is TRUE, the routine returns NULL (immediately) if the message *which* is not on the process' queue. Otherwise the message is removed from the queue and returned as the (non_NULL) result.
If *which* is NULL and *poll* is FALSE, the process is suspended until any message is placed on its queue. The (head) message is removed from the queue and returned as the result.
If *which* is NULL and *poll* is TRUE, the routine returns NULL (immediately) if the process' message queue is empty. Otherwise the head message is removed from te queue and returned as the (non-NULL) result.

Non-NULL use of the *which* parameter produces a synchronous request/response model, otherwise the process is essentially event-driven. The *poll* parameter allows a process to perform its own scheduling by checking explicitly for new messages.

rome_fatal

void *rome_fatal*(
  char **why*)

The *rome_fatal* routine is the last-ditch error handler for a ROME system. It prints the error message onto the default UART, using *rome_kprintf* then enters the system debugger with all interrupts disabled.

rome_find_queue

ROME_PROCESS *rome_find_queue*(
  const char **name*)

The *rome_find_queue* routine returns a handle to a process' queue, given a string name for the process. The handle is used as the destination field for messages sent to that process. Handles are stable and need only be obtained once per queue.

rome_free

void *rome_free*(
  ptr *memaddr*)

The *rome_free* routine returns a block of memory pointed to by *memaddr*, previously allocated by a call to *rome_alloc*, to the free pool. Memory is returned in the units in which is was allocated (i.e. it is not possible to free half of an allocated block).

rome_kprintf

void *rome_kprintf*(
  char **format*,
  . . .)

The *rome_kprintf* routine is a simple formatted print routine intended to be used from interrupt contexts or error situations where *printf* is otherwise unavailable. The routine runs as a critical section (to ensure the output appears uninterrupted) and uses the polled-mode UART *serial_out* routine to output characters according to the supplied format. The only formatting options supported are: ‘d’ for integers; ‘I’ for IP addresses; ‘x’ for hexadecimal; ‘c’ for character; and ‘s’ for string. A length option may be given to ‘d’ and ‘x’ It is ignored for ‘d’ but implemented for ‘x’. The string ‘%%’ prints a single ‘%’ character.
rome_kprintf is not intended for normal output. The system clock will pause while output is being produced and characters from normal processes may be lost In general, use of *rome_kprintf* should be limited to the ‘init’ routines of processes (before the scheduler is started), error messages from interrupt handlers (which do not have access to standard output files) and calls to *rome_fatal*.

rome_reply

(void) *rome_reply*(
  ROME_MESSAGE **_m*)

The *rome_reply* macro returns a message to its source. It sets up the reply fields from the source fields and calls *rome_send_message* to enqueue the message to the original sender process.

rome_send_message

void *rome_send_message*(
  ROME_MESSAGE **msg*)

The *rome_send_message* routine directs a message to the destination process The fields of the message structure have already been initialised, including the source and destination queues, and the message operation. The routine calls the process’ queue handler (if one is registered) and either returns the message to the source if it has been handled completely, or queues it for the process.
This routine also implements most of the core scheduling policy. If context switching is allowed at this point (i.e. *rome_send_message* is not being called from an interrupt handler), and the message priority is higher than the current process’ priority, and the destination process is able to accept the message (e.g. it is not blocked waiting for another, specific, message) then a context switch will occur directly. Otherwise the scheduler queues are updated to reflect the changed state.

rome_start

void *rome_start*(**void**)

The *rome_start* routine performs the machine-independent intialisation of the ROME system. It calls the cpu-dependent initialisation routines, initialises the memory manager, and calls the ‘init’ routines for all processes in the system as it sets up the process tables. The routine exits by calling the system scheduler to perform a context switch to the first runnable process.