An Example Design of a Non-preemptive Real-Time Kernel

Version 1.0

This is a draft of the document. It may be missing information and is incomplete. Please report any errors or omissions.

P. Dasiewicz © 2009

Introduction

This version of the document is not meant to function as a standalone all inclusive beginner's guide to the design of an embedded real-time executive (the kernel). It is to accompany the RTX lecture slides and it is assumed that at least the first 3 chapters of the course text have been covered in lectures. A possible design for a basic non-preemptive message passing RTX is initially presented and later extended to account for process preemption. Possible pseudocode is provided as implementation examples (as appropriate) and many course project related issues are discussed but without detailed solutions – which would be expected to be solved during the the course project. Later sections overview the gcc / MCF5307 run-time environment with detailed examples of combining assembly language with C-statements and using the gcc run-time stack frames to advantage.

Table of Contents

1 RTX Functional Overview

This document is a reference text to RTX lecture slides. First, we overview the basic desired functionality and any limitations of our basic non-preemptive real time executive. In later sections, the basic design is extended to include preemption.

1.1 Process Management

At the very least we need to be able to create and terminate user processes. Since we are considering an embedded application, the ability to terminate a user process is not needed since the assumption is that all processes exist for the lifetime of the system once they are created. A second requirement is to perform a complete system initialization at system start (power-up). This would include initialization of all hardware, creation of all processes, initializing the required stack and heap spaces.

1.2 Processor Scheduling

We need to select a process that is ready to execute and change it to executing. A criteria must be defined to chose among many ready processes.

1.3 Interprocess Communication and Synchronization

Some processes need to communicate and synchronize their actions with each other to perform some overall functionality. The communication and synchronization mechanisms must be defined.

1.4 Storage Management

A mechanism to allocate and deallocate memory requests must be provided. This mechanism will be customized to the requirements of the intended application to reduce the overhead costs.

1.5 Interrupt Handling Framework

Capture interrupts and, if required, start (or make ready) a user process that was blocked waiting for the event to occur.

1.6 Timing Services

Since real-time is the application, the RTX must provide services which measure time or occur at specific points in time. There may be a need for either (or both) absolute and relative timing requests.

1.7 Device Driver Interfaces

There is need to provide standard I/O and specialized interrupt driven device handlers.

2 Simple RTOS: Requirements

To reduce complexity, the following basic requirements will drive the design. Initially, the assumption is the RTX is to be non-preemptive. This constraint is initially imposed to reduce the initial complexity of the design and will be removed near the end. Since the intended application is embedded real-time, the requirement of process creation after initialization is not needed. Also, no process terminates during the lifetime of the RTX. These two

requirements greatly simplify the design. In a general purpose OS, these requirements would not be appropriate and never made.

We assume that the priority (process urgency) is sufficient to use as a scheduling criteria. Further, process priority is static and does not vary dynamically.

The IPC mechanism is to be communication via message envelopes (i.e. message passing). Processes will send and receive message envelopes (shared blocks of memory) to communicate and synchronize their actions. Sending a message will be asynchronous while receiving a message will be synchronous.

A simple form of memory management will be used. Fixed size blocks of memory will be allocated/deallocated by processes and used as message envelopes for IPC. This avoids the extra complexity and overhead of variable sized memory allocation/deallocation.

There is the need for only a basic form of timing services: relative time.

Some additional assumptions which greatly simplify the RTX complexity include the following:

- All processes are known and created at initialization time. This removes any requirement to dynamically create processes.
- All processes are “friendly, co-operating and non malicious”. This produces considerable complexity reduction since most 'security' related issues can be ignored. The RTX trusts each process to do the right thing and user processes will never do anything that could harm the RTX or each other. No general purpose OS would ever make such an assumption. For example, there is no need to track memory allocated to a process since the process will always return it. Since only a non-preemptive design is initially considered, this would also imply that a process would voluntarily yield the processor at designated points in its code in order not to monopolize the processor.
- For a process to send a message to another process, the sender must know the process id of the receiving process. The assumption is that all processes know the process id of all other processes and hence no RTX primitives need be provided for a process to determine its own process id or that of any other process.

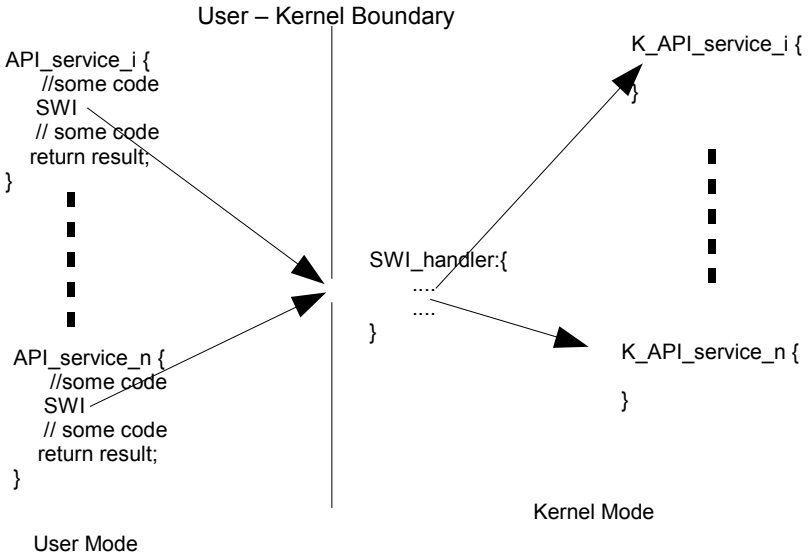
3 The User Process / Kernel Interface

A user process executes in a restrictive mode (internal kernel data structures are not accessible or visible, usually with only a subset of the CPU instructions available, a restricted address space and often with a separate user stack,...). This execution mode is usually referred to as *user mode*. The kernel executes in the CPU's *supervisor mode* (also referred to as kernel mode) and can execute all the CPU's instruction set and has access to the complete address space with no restriction on its operation. When a user process is scheduled (by the kernel) to execute, then the kernel ensures that the user process only executes in *user mode*.

The kernel provides a set primitives (services) to user processes. A user process is provided a user visible API. The API defines all services (and their signature) that the kernel provides. As seen in later sections, the actual service (functionality) of each API service must be executed in kernel mode. Thus, the initial problem is to determine the actual user process / kernel interface. What happens when a user process invokes an API service? Since the API defines only user visible services, the actual function invoked is still in user mode and cannot by itself provide the required service since the service can only be run in supervisor mode , that is, kernel mode.

Fundamentally, in response to the user process' call to a visible API service, the service performs the following operations: i) repackages the passed parameters, ii) enters the kernel by a software interrupt, iii) extracts any return value from kernel, and iv) performs a return to the user process. Usually, the kernel has a single entry point which is entered by issuing a software interrupt (SWI) instruction. This causes an exception handling sequence to be performed by the CPU and control is transferred to the SWI handler provided by the kernel. The SWI handler (executing now in kernel mode) will determine which kernel service was requested and perform the service in kernel mode. At the completion of the service, control is transferred back to the user visible API service and it resumes execution in its original user mode. See Appendix A for a detailed example.

The following figure illustrates the basic user process API to kernel interface. The kernel version of the user visible API is preceded by a "K_API_service_i" to indicate that this is the actual kernel implemented service. A short example will illustrate the mechanism.



The following generic example is based on the following assumptions:

1. No special knowledge of the C compiler , its run-time stack frames or assembler is assumed.
2. No particular processor is assumed except that it can perform SWI's; and that it has defined data registers named D0, D1, ...
3. On a SWI, the D0 register will hold a constant denoting the required kernel primitive. And on return from SWI, the register D0 will hold any returned value.
4. Assume that the function ASM("*processor instruction*") is known to the compiler and will cause it to place the processor instruction into the produced assembly language at the given point.

Consider the following user visible API function for a send message primitive (discussed later):

```
int send_message( int dest_pid, char * msg )
```

This API is a request to send a message to a process with process id *dest_pid* and it returns an integer result.

A generic C implementation of send_message at the user level could be:

```
int send_message( int dest_pid, char * msg)
{
      int result_value;

      ASM( "PUSH D2");          // save on stack some registers
      ASM( "PUSH D1");
      ASM( "PUSH D0");
      ASM( "MOVE dest_pid, D1");
      ASM( "MOVE msg, D2");
      ASM( "MOVE #K_SEND_MSG, D0");  //the SWI handler will look at D0
      ASM( "SWI" );                   //enter kernel here
      ASM( "MOVE D0, result_value"); // save the returned value
      ASM( "POP D0");           // restore registers from stack
      ASM( "POP D1");
      ASM( "POP D2");
      return result_value ;
}
```

The generic example illustrates the basic outline used for all user visible API functions. This can be optimized once the C compiler and its run-time stack frames are known. Other user visible API functions would be implemented as above with the basic difference being the number of parameters in the API function itself and the return type of the result.

We can now determine a generic outline for the kernel's SWI handler. All user API's perform a SWI to the same handler. The SWI handler determines the nature of the request by examining the D0 register which would have a different value for each user API. The following illustrates a basic generic SWI handler:

```
SWI_handler:
{
      int int_var1, int_var2, ret_value;
      char * ptr_1;
      char * ptr_2;

      // some initial code to save appropriate registers, etc....
      // all done in assembly language (also disabling interrupts?)
      ASM("MOVE D1, int_var1"); //copy the dest_pid from user
      ASM("MOVE D2, ptr_1");    // the ptr to the message from user
      ASM("MOVE D0, int_var2");  // copy the service identifier
      switch( int_var2 ) {
            case  K_SEND_MSG : // the send message was invoked!
                               ret_value = K_send_message( int_var1, ptr_1);
```



```
                                    break;
                case other_selections :  // other kernel services as needed
            };  //end off the switch statement
            // Now restore any saved registers and prepare for return to user.
            // Add any assembly language here to restore registers saved earlier.
            // put the return value into register D0 for the user API
            ASM("MOVE ret_value, D0");
            ASM( " RTE "); // the rte will now restore the state of the processor
                           // back to the statement following the original SWI
}
```

The entry to SWI handler 'appears' to be customized to the send_message API, but in general, it will work for all user API calls which have at most 2 formal parameters. The actual type conversion from the entry code to the actual parameters required by each kernel API function will be performed individually (perhaps using 'casts') by the respective 'case' alternative of the 'switch' statement. On return to the user API function (after the SWI), register D0 will have the appropriate return value and this will be returned to the user process (as seen previously).

4 Requirement for Atomicity

There is a requirement for each RTX primitive to be atomic, that is all instructions of any primitive must execute indivisibly with no interruption. Once a kernel primitive starts, it runs to completion with no interruption. This is required since primitives access and modify private kernel variables and data structures. It would not be permissible for an interrupt handler to access/modify the same kernel data structures concurrently with a primitive's execution. Thus, it must be ensured that all (or least any potentially interfering) interrupts be disabled during the execution of the primitive's code.

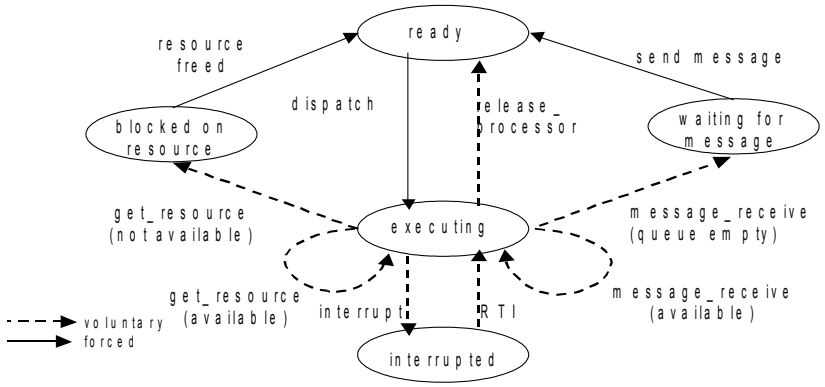
In reality, all the primitive functionality will be performed in “kernel mode” (with interrupts disabled) by kernel versions of the user visible API functions, after the user API ‘traps’ into the kernel. On entry, the software trap (SWI) handler ensures that interrupts are disabled while the kernel primitive is executing.

Since the primitives (services) run in kernel mode (after a SWI from the user visible API), it is simple to disable all (or some potentially interfering) interrupts by the SWI handler. Recall that in kernel mode, it is now possible to execute privileged CPU instructions to disable/enable interrupts. After the service completes, then the original interrupt masks (used in user mode) can be restored prior to returning to the user process.

All kernel primitive implementations, in following sections, assume that execution is in kernel mode and that atomicity is guaranteed.

5 Process State Diagram

Since a process can be in various states during its execution, the process state diagram is required. The following is a typical state diagram that can be used in this example design. While in the executing state, a process can voluntarily yield the processor (via the 'release_processor' primitive) and return back to the ready state allowing another process to be selected for execution. If a process requests a message_receive and no message is available, then the process blocks and is placed into the waiting_for_message state until another process sends it a message; after which it returns to the ready state. If a process requests a resource (get_resource primitive) and no resource is available, then the process blocks and is placed in the blocked_on_resource state until a resource becomes available. The dispatch action of the kernel selects a ready process for execution and changes its state to



executing. A process moves from executing to interrupted state during interrupt handling and returns to executing state after the interrupt handler completes. Since our initial design is non-preemptive, a process always returns to executing from the interrupted state. For a preemptive system, some changes would be required to the state diagram.

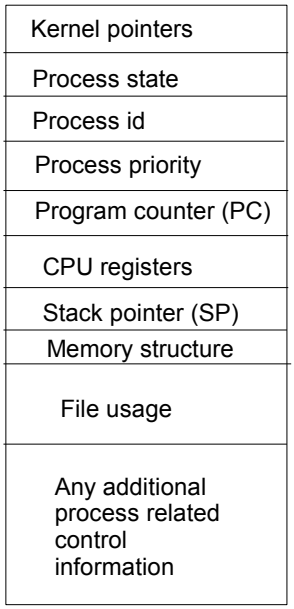
6 Kernel Private Data Structures/Functions

Some of the private kernel variables, data structures and functions are examined. These structures are not visible nor accessible by a user process.

6.1 The Process Control Block (PCB)

In order to manipulate a process (i.e. make it ready, block it, put a process on a queue, etc), the kernel needs an internal representation of a process. A data structure that contains process related information is required. It is this data structure that is placed on linked lists etc. not the physical process itself. Typically, a process control block (PCB) or a process object is used to represent a process. The PCB is not accessible to a user process since it is a private data structure maintained by the kernel. The following figure illustrates a typical PCB data structure which can be used. Depending on the application, the PCB can contain many different fields, some of which are discussed.

Each process (user process, system process or i_process) requires a PCB which is constructed during process creation and remains in existence until the process terminates. The PCB contains several kernel pointer fields allowing the PCB to be placed on one or more linked lists simultaneously. For example, each PCB is permanently placed on the system process list which is a linked list of all processes in the system. In addition, the PCB may also be placed on various blocked queues and ready queues.



The "process state" field is used to indicate the current state of the process (i.e. ready, blocked-on-receive, executing, suspended, ...).This field is maintained by the kernel during the lifetime of the process. Kernel primitives examine and modify this field, e.g. when a process is made ready or is blocked.

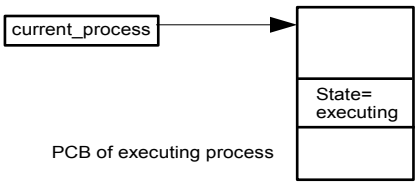
The "process id" field holds a unique identifier for each process. This is usually a unique integer number assigned to each process and not reused again. The "process priority" field contains the priority or urgency of a process and is used to schedule processes and to determine if process preemption is required. This is usually an integer in some restricted subrange.

The next three fields "PC, CPU registers, SP" are used to maintain the process context when the process is not executing. The process context (CPU registers) can be saved here when the process is not running and can be used to restore a process to a running state. These fields are only valid when a process is not running and its context has been saved.

The memory structure field can be used to maintain information about memory allocated to the process (for example in a VM environment). The "file usage" field can be used to record which files have been opened by the process and the state of each file. Additional fields can be used to maintain various statistics required by the kernel (total execution time of process, waiting time, number of page faults, etc...).

6.2 The *current_process* Variable

The kernel must know the identity of the currently executing process at all times. For example, when a process invokes a kernel primitive, the primitive must be able to deduce the identity of the requesting process; also an interrupt handler might need the identity of the process that was executing at the time of the interrupt. The kernel maintains a variable called `current_process` which points to the PCB of the currently executing process (see adjacent figure). This variable is updated each time a different process is chosen for execution.



6.3 The *process_switch*() Function

This is a frequently needed procedure to i) remove the current process from the CPU, ii) select the next process to execute, and iii) transfer control to the selected process (restart it). For example, an RTX primitive decides to block the requesting process and to select another process to run. Alternatively, in a preemptive system, an

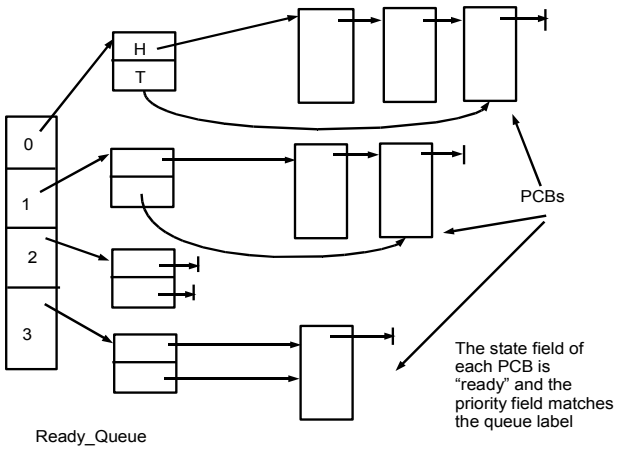
interrupt handler makes ready a process of higher priority than the currently executing process, and decides to switch to the newly ready'ed process.

For example, before calling `process_switch`(), a primitive would have placed the blocked process on a blocked queue (if appropriate) and made any needed update to the state field of the current process's PCB. No return to the call is made until the original process is restarted again in the future. In this example, process switch would call the scheduler to select a new process to run. Then, a call is made to the function `context_switch(new_pcb_ptr)` passing to it a pointer to the PCB of the newly selected process to execute. Context_switch performs the following: i) saves the context of the currently executing process (if not done already), ii) sets the variable `current_process` to refer to `new_pcb_ptr` , iii) sets the state of the new `current_process` to executing, iv) restores context of `current_process`, and v) restarts the execution of `current_process`.

After a primitive invokes `process_switch`, the invoking process will eventually regain execution again and will restart on the statement following the call to `process_switch`. See a later section for a more detailed look at this functionality.

6.4 Process Scheduling

The requirement specified a fixed priority based approach where each process is assigned a priority (a process urgency). The highest priority ready process will be chosen to execute. Processes with equal priority are treated as first-come-first-served (FCFS). This priority based scheduling is applicable to both pre-emptive and non pre-emptive applications. As with any fixed priority scheduling, there is always a danger of starvation for the low priority processes if there are too many higher priority processes ready to run. Other scheduling algorithms are used in more complex real-time systems. The following figure shows a 4 level priority ready queue with the highest priority level being '0'.



6.4.1 The *rpq_enqueue/dequeue* Functionality

There is also need of enqueuing and dequeuing PCBs to/from the ready process queue (for example, the previous figure). `rpq_enqueue(pcb_ptr)` enqueues the specified PCB onto the appropriate ready queue by obtaining the process's priority from the priority field in the PCB and inserting the PCB to the end of the respective queue.

`rpq_dequeue()` is a function that returns a pointer to the PCB of the highest priority ready process. The PCB is also removed from the respective queue. The function scans the four queues (queue 0 is the highest priority) and

determines the highest priority ready process that is to be run next and removes its PCB from the respective queue.

6.4.2 Null Process

What happens if the scheduler cannot find a ready process? This could occur if all the ready queues are empty. We cannot just wait inside the kernel for some process to become ready. Since all processes appear to be blocked, then either the system is in deadlock or one or more processes are waiting for some I/O device to complete and the appropriate interrupt handler might make a blocked process ready and save the day. This condition must be accounted for in the design. There are two fundamental alternatives in this case.

First, we could intelligently loop inside the kernel waiting for an interrupt to make a process ready. For example consider the following pseudo code:

```
while (ready_Q is empty ) do
    EI              //enable interrupts
    DI              // disable interrupts
endLoop
```

The loop enables interrupts for a brief time period such that if there is a pending interrupt, then the processor will recognize the interrupt, perhaps resulting in an interrupt handler making a blocked process ready. With this approach, the interrupt handlers must be aware that the kernel is in this loop when the handler runs otherwise the handler might force a process switch with some unforeseen consequences. This approach is not recommended for the project although there are some beneficial results in this approach from the performance point of view.

An alternative approach might be to include a special *null process* that is always ready to run and never blocks on any kernel primitive. The null process's priority is set to the lowest priority level such that it should not be selected by the scheduler unless there are no other ready processes. When the ready queue is "empty" of user processes, then the scheduler would pick the null process to execute next.

A simple implementation of the null process (for a non preemptive system) could be as:

```
null_process( )
{
       while ( true )  {
            release_processor( );
       }
}      //end of null_process
```

The null process consists of an infinite loop with a call to the `release_processor( )` API service. The `release_processor` service is equivalent to a "yield" request indicating that the executing process is willing to yield to another process of equal or higher priority. The null process executes in user mode. If a user process is made ready by an interrupt handler during the null process's execution, then the yield request will cause a process switch to the ready process with the null process returning to the ready queue. If no other user process is ready, then the null process will resume execution and continue in its loop.

In a preemptive system, the null process could be reduced to the following:

```
null_process( )
{
       while ( true )  {
       }
}      //end of null_process
```

The call to `release_processor( )` is not required since if an interrupt handler makes a user process of greater priority ready, then the interrupt handler will also perform a process switch to the higher priority process and place the null process back onto the ready queue. This will also improve the response time to switch to the user

process that was made ready by the interrupt handler.

The null process could also perform some useful function, such as a diagnostic function, for example to compare checksums in ROM, etc. The first restriction for a null process is that it makes no API calls that could block it. In a non-preemptive system, a further restriction is that time duration between successive calls to `release_processor()` be short enough to allow an acceptable response to external interrupt generating events. This additional restriction is not needed in a preemptive system.

6.4.3 The *release_processor* Primitive

The null process requirement led to the need for a `release_processor` primitive to be added to the RTX. A user process calls the `release_processor()` primitive to voluntarily yield to another process (i.e. the currently executing process is requesting the kernel to switch to another ready process of equal or higher priority). If there is no such ready process, then the calling process will continue to execute. The basic pseudocode for `release_processor` can be given as:

```
K_release_processor():
{
        set current_process state to ready
        rpq_enqueue(current_process)
        process_switch()

}
```

`release_processor` just places the currently executing process back onto the ready queue, then picks the next ready process and performs a process switch to the selected process. If there are no other ready processes, then the caller gets picks again and continues to execute. The pseudocode could be optimized to reduce any latency.

6.5 System Initialization

At OS start up (e.g. power up reset), many initialization operations need be performed such as hardware initialization, OS data structure construction, process creation, selecting the first process to run. In a dedicated embedded application, the OS must know which and how many processes to create. Our design will include an array of records (known as the Initialization Table (IT)). Each record entry contains all the information necessary to start its respective process. A record, for process_i, could have the following simple structure:

Record_i

process_id
process priority
required stack size
initial PC

During initialization, the IT would be processed and the following operations would be performed for each table record (process_i defined):

- allocate a new process control block (PCB) for the process
- initialize the PCB with the process_id, priority and initial PC obtained from the IT record
- allocate a block of memory from the stack space based on the requested stack size and place the value of the processor's SP into the PCB
- enter the process's initial PC into the PCB (this is the start address of the process code)
- since the process has never executed before, there is no previous context for this process. Create an initial context based on the SP and initial PC and save it into the PCB or to some other context save

area for this process. Recall that a context includes the values of all CPU registers, status register (SR), etc. This initial context will be used with a 'RTE' (return from exception) instruction to start the process for the first time. Hence a previous context must be created for the RTE instruction.

- Set the state field of the PCB to "ready"
- enqueue the PCB onto the ready queue in its proper position based on its priority.

As discussed in a later section, *iprocesses* will be handled with a slight difference in regard to the process status field. Once the entire IT has been processed, all internal kernel data structures created and all hardware initialized then the scheduler is invoked to pick the highest priority ready process and this process is started into execution.

7 IPC Primitives

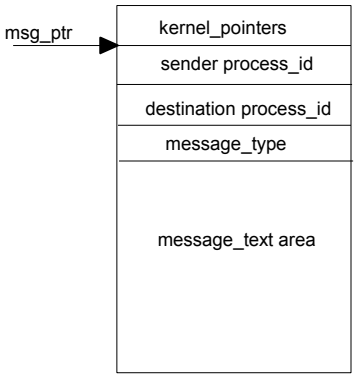
The design of several basic primitives is given. These primitives are only a sample of the required primitives and give an example of the design constraints and possible solutions. They are based on a basic non-preemptive system with some insight on the conversion to a preemptive system.

7.1 Inter-Process Communication (IPC)

In general, processes do not exist in isolation, they need to communicate with other processes and to synchronize their actions. The requirements call for a message-based IPC scheme. Messages are carried in shared memory blocks. A process writes a message into a shared memory block, sends a pointer to the memory block to another process and the receiving process reads the message from the memory block. The shared memory blocks are called "message envelopes". The sender of a message would invoke the RTX send message primitive supplying a pointer to the message envelope and the process id of the recipient. The immediate issues are what is the format of the message envelope and where do message envelopes come from.

Message envelopes are managed by the kernel. To avoid run-time overheads, some fixed number of them are created during system initialization. Basically, this is a very simple form of memory management of fixed length memory blocks. A process would allocate a message envelope to send a message and deallocate an envelope when no longer required. The assumption is that a process owns a message envelope that it allocates or receives until it sends or deallocates the message envelope.

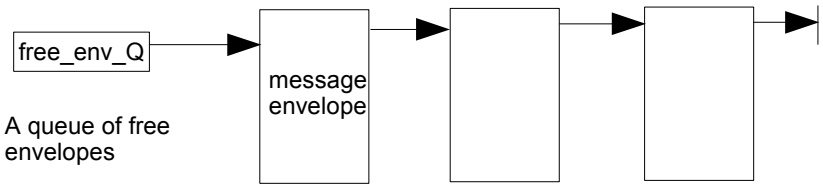
First consider the design of a message envelope and its internal structure. The following figure illustrates a message envelope as a fixed length memory block with a default number of fields. The `kernel_pointers` field contains one or more pointers used by the kernel to place the message envelope on various linked lists. Ideally, this field should not be accessible to a user process. The next two fields hold the process_id of the sending process and the process_id of the destination (receiving) process. Typically, the kernel would update these fields whenever a message envelope is sent to another process. The `message_type` is an optional field which can indicate the type or class of the message to the receiving process. The actual text of the message is written into the message_text area by the sender. A receiving process can examine the sender process_id field to determine the sender of the message. Additional fields can be added as required.



7.2 Message Envelope Management

The question is where do message envelopes come from. We assume that a fixed number of message envelopes are created at initialization. This prevents any run-time overhead of allocating blocks of memory to be used as message envelopes on demand. The fixed number of message envelopes are initially placed on a free envelope

queue as illustrated below. The kernel_pointers field is used to form a linked list of unallocated message envelopes.



The kernel uses the variable `free_env_Q` to hold a pointer to the first free envelope. If a process does not have an envelope then it allocates an envelope first. If there are no free envelopes available, then the process blocks and must wait until an envelope becomes available (i.e. deallocated by some other process). The blocked process can be placed on a blocked for envelope queue joining any other processes waiting for a free envelope as shown in the following figure. The PCB of the blocked process is enqueued onto the blocked queue either in FIFO order or the queue can be sorted according to process priority (as appropriate). When a message envelope becomes available, then a PCB can be dequeued from the blocked queue and be given the available message.

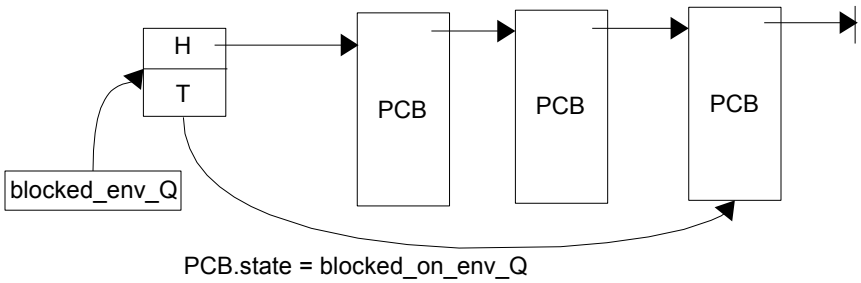
7.3 Message Envelope Allocate

To obtain a message envelope, a process calls the `allocate_envelope` RTX primitive. If there is a free message envelope, then the process is returned a pointer to the free envelope. If there is no free envelope available, the process blocks and is placed on the blocked envelope queue and a process switch occurs. As an example, consider the following pseudocode for allocate_envelope (where MSG is a type specifier for the message envelope):

```
MSG * K_allocate_envelope() {

       while (free_env_Q is empty) {
              put PCB on blocked_env_Q
             set process state to blocked_on_env_allocate
              process_switch();
                    // restart here when blocked process executes eventually
          }
          env ← reference to de-queued envelope
       return env;
}
```

The “while” loop is used to allow the kernel to retry its check on the availability of a free envelope even if the process becomes ready. For example consider the following scenario. A process blocks on its attempt to obtain an



envelope. Sometime later a different process deallocates an envelope, and the blocked process is made ready and

placed on the ready process queue before it can try again (due to the process switch). While it is on the ready queue, another executing process might possibly allocate the currently free envelope since it was not given to the blocked process. Then when the process attempts to check the free envelope queue, it might find it empty causing it to become blocked once again, and so on. As will be seen later, this sequence can be optimized once the deallocate envelope primitive is discussed. In this example, the allocate_envelope primitive only makes a blocked process ready, put on the ready queue and is given the opportunity to try again to obtain an envelope. There is no guarantee that a free envelope is still available once the blocked process restarts after the call to process switch. So, the primitive works, but could be made better.

7.4 Message Envelope Deallocate

A process calls `deallocate_envelope` to return an envelope it no longer requires. The freed envelope is placed on the free_envelope_q. The blocked queue of processes waiting for envelopes is checked. If the queue is empty, then the primitive finishes. Otherwise, a blocked process is removed form the blocked queue, made ready, placed on the ready process queue allowing the process to retry its allocation again. The following pseudocode, for a non-preemptive system, illustrates a possible implementation:

```
int K_eallocate_envelope( MSG *env ) : {

       put env onto free_env_Q
       if ( blocked_env_Q not empty)
         { dequeue one of the blocked processes
            set its state to ready and enqueue it on ready process queue
          }

}
```

The primitive does not block the calling process. The freed envelope is not directly given to a waiting process, but rather the process is made ready and allowed to retry its allocation attempt. This can be improved by providing a mechanism to give the freed envelope directly to the process made ready and then modifying the allocate primitive accordingly. This is left as an exercise.

Although the IPC primitives could be improved to immediately deliver the freed message envelope to a waiting process, consider the following scenarios where we assume a preemptive system and the blocked on envelope queue is sorted by process priority:

1. If deallocate envelope makes a waiting process ready and the ready'd process is of higher priority than the process performing the deallocate, then an immediate process switch will occur and by default the newly ready'd process will obtain the envelope.
2. If deallocate envelope makes a waiting process ready and the ready'd process is of lower or equal priority than the process performing the deallocate, the ready'd process will wait on the ready queue until it is scheduled. No process of lower priority can allocate the newly freed envelope. But since the ready queue is FIFO ordered when processes are of equal priority, another process (ahead of it in the in the queue) of equal priority could allocate the newly freed envelope. This makes the primitive implementation seem less fair since the original process might possibly block a second time when it is scheduled.
3. If deallocate envelope makes a waiting process ready and the ready'd process is of lower or equal priority than the process performing the deallocate, the ready'd process will wait on the ready queue until it is scheduled (a further continuation of scenario 2). Next, a higher priority process is either scheduled to run or made runnable by an i_process. The higher priority process can now allocate the newly freed envelope before the original process. This may seen unfair to the original process but it

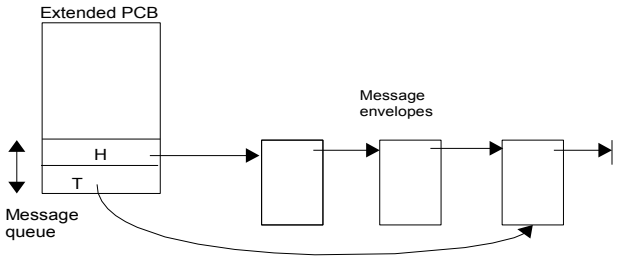
does follow the process priority objectives and may even reduce effects of priority inversion.

7.5 Message Receive

The requirements call for a synchronous message receive. An initial design issue is how does the kernel buffer messages that are sent to a process but the process does not do a receive for some time. A simple solution is to have the kernel maintain a message queue for each process and place the queue head/tail structure in the process's PCB as an extra field. The following figure illustrates how the PCB is extended to perform this additional functionality. Each PCB now has a new message queue field to hold messages sent to a process but not yet received.

The queue could be implemented as either FCFS or be sorted based on the priority of the sending processes.

If a process does a receive and there are no messages waiting for it, then the process blocks with its status set to



blocked_on_receive.

The following pseudo code illustrates a possible implementation of the receive message primitive:

```
MSG * K_receive() {

   if ( current_process's msg_queue is empty) {
         set state of current_process to blocked_on_receive
         process_switch( );
       //*** return here when this process executes again
   }
   env ← dequeued envelope from the process' message queue
   return env

}
```

If the process's message queue is empty, then the process blocks. Eventually the process will restart once a message is sent to it. At that time, the process is returned a pointer to a message envelope.

7.6 Message Send

Since the message send is asynchronous, the sending process does not block on a message send. The message is delivered to the message queue of the destination process. If the destination process is currently blocked waiting for a message, then the process is made ready and placed on the ready process queue. The following pseudo code is an example of a possible non-preemptive implementation:

```
int K_send( int target_pid, MSG *env) : {

   set sender_procid, destination_procid fields in env
   target_proc ← convert target_pid to process PCB reference
   enqueue env onto the msg_queue of target_proc
   if ( target_proc.state is blocked_on_receive)
     { set target_proc state to ready
```



```
            rpq_enqueue( target_proc );
          }

      }
```

7.7 Possible Extensions for *send/receive*

Although the original requirements did not specify other forms of send/receive, it is relatively easy to add additional forms. For example an asynchronous receive can be added to return, say a null pointer, if there is no message waiting for a process (this is done for the special case of an i_process calling receive). Also, a synchronous send can be added to block a sender of a message if the destination process is not already blocked waiting for a message and then make the sender ready when the message is finally given to the destination process when it invokes a receive.

In addition, we could add a receive which specifies the expected sender of the message. Here, several semantics are possible. First, a null message pointer could be returned if a message from the specified sender is not already waiting. An alternative could be to give 'priority' to messages of the specified sending process if there are multiple waiting messages.

7.7.1 Prioritized Message Queues

Messages waiting on a process's message queue could be enqueued either in simple FIFO order or be sorted according to priority. If it is chosen to sort the messages according to priority, then the next design problem would be the definition of 'priority' from the perspective of the message envelope. Does a message inherit the priority of the sending process or is the priority encoded into the message itself regardless of the priority of the sender. Either definition (or both) could be used depending on the application.

7.7.1.1 Inherited Message Priority

Consider the case of a message inheriting its priority from the sending process. This could be valid when a high priority process sends a message to a lower priority process. Thus when the lower priority is eventually scheduled, it is assured that it will receive messages from any higher priority processes before any other messages are received. This is also valid when high priority processes send messages to each other. One obvious consequence is a possibility of starvation of the lower priority messages (from low priority processes) if the higher priority processes are very active in sending messages to each other. If the message traffic is mostly between processes of 'equal' priority, then the inheritance based priority sorted queue is of little advantage.

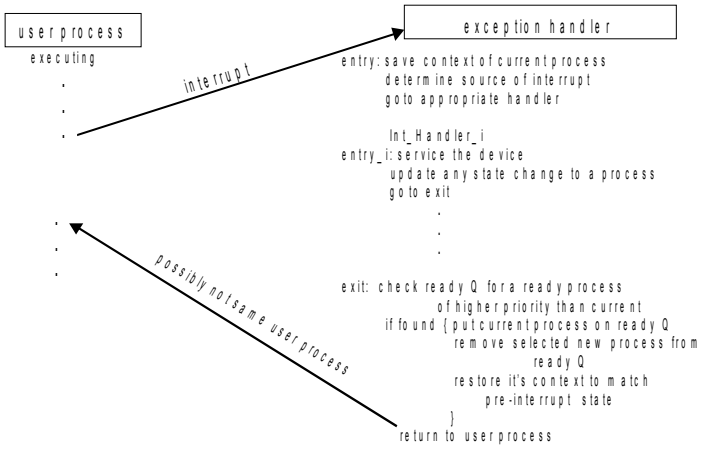
Another design issue is how its priority is maintained by the message while enqueued on the message queue. Its priority will need to be examined each time a new message is enqueued. Currently, the message envelope has no capability to maintain priority information. This can easily be fixed by adding another field to the message envelope called, say '`msg_priority`' . The modified send primitive would copy the sender's priority (from its PCB) into the message envelope's *msg_priority* field prior to placing the message envelope into the priority sorted message queue of the destination process. This would be a permanent addition to the fields of a message envelope.

7.7.1.2 Priority based on *message type*

It might be possible to imply the priority of a message based on its message type. The problem is that in many situations, the message type has only meaning (or significance) to the processes exchanging messages of a particular set of message types. Thus, the kernel (or send primitive) would need to be aware of the sets of communicating processes and understand the meaning of the message types in order to infer priority. This could be done by defining a fixed set of global message types in some priority order. But this does not seem to be very attractive or desirable. If done, then sorting of the messages can easily be performed by examining the existing message type field of the message envelopes.

8 Interrupt Handling

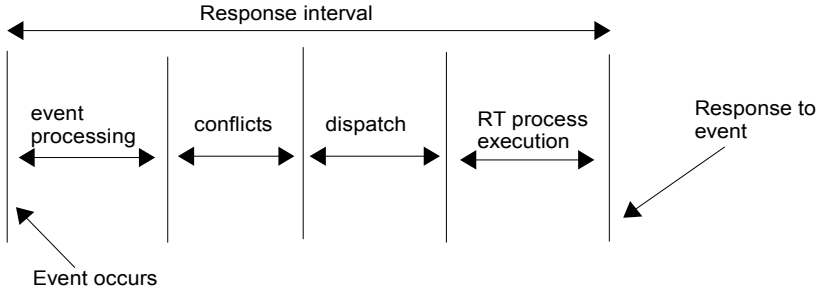
In a real-time system, interrupt handling is one of the more important design issues since the latency in response to an interrupt must be at a minimal and requires efficient and speedy interrupt handlers. Interrupt processing may cause a blocked process to be made ready and placed on the ready process queue. For example, a process blocked for external event to occur will have its state changed to ready and be placed on the ready process queue when the event occurs. In a preemptive system, the user process executing at the time of the interrupt could be preempted by a higher priority process made ready by the interrupt handling. A possible interpretation of an interrupt is a hardware message requiring a short response latency and short service time.



Design issues include: does the interrupt handling code run as part of the kernel, within a process (if yes, which?); are interrupt handlers themselves interruptible; if the OS is preemptive, need to deal with the possibility that an interrupt results in a higher priority process becoming ready and performing an unscheduled process switch.

The adjacent figure illustrates a generic interrupt or exception handler operation for a preemptive system. After the servicing the interrupt, the exception handler checks to see if a higher priority process was made ready by the service. If yes, then a context switch is made to the newly ready'd process and the original process running at the time of the interrupt is placed on the ready process queue. The figure depicts the situation where all interrupt sources (devices) are 'OR-tied' to a single CPU interrupt line. Thus, the initial interrupt handler determines the device that caused the interrupt and then transfers control to the specific handler for the device.

One of the more important design issues involves the latency (or the elapsed time) between the occurrence of the interrupt and when the real-time process that is waiting for the interrupt is actually run and has performed the necessary actions related with the interrupt. The following figure illustrates the situation.

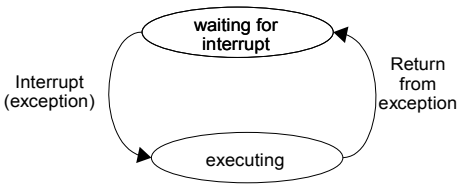


When the event occurs, it requires some initial event processing to make the RT process ready and place it on the ready process queue for scheduling. There may be conflicts since it may have to wait its turn before being chosen for execution. There is some dispatch latency to select the process, restore its context and start its execution. Finally, the RT process needs execution time to complete the event processing. One goal is to make the response interval as short as possible by reducing the event processing, conflicts and dispatching overheads.

8.1 Concept of an *i_process*

In general, the interrupt handler must interact with the real-time process that will be performing the event handling as discussed in the last section. There are many ad-hoc mechanisms that could be employed to reduce the response interval. Let us consider a structured approach using the concept of an *i_process*.

The i_process gets the CPU through an interrupt handling sequence, not through the dispatcher. It never blocks if it invokes a kernel primitive (the primitives must be aware that an i_process invoked the primitive). The interrupt (exception) handling routine starts the appropriate i-process. Conceptually, the i_process has the highest priority and is scheduled (started) by the interrupt handler. The state diagram for an i_process is shown. The i_process can execute in kernel mode which may give it special privileges not available to the basic user level process.



i_process state diagram

When idle, the i_process is always ready to run but not placed on any ready queue. The i_process has an associated PCB with the state field permanently set to "`i_process`". An i_process can invoke any kernel primitive, however, an i_process is not allowed to block! Primitives which can block a process must be modified to ensure that an i_process does not block! For example, the synchronous receive message primitive should be modified to return null if the invoking process is an i_process and there is no message waiting. Similarly for other primitives which can potentially block a process or perform a process switch. If the design of an i_process requires that it send messages during its execution then it must be ensured that the i_process has a sufficient supply of messages or can get them without invoking the allocate_envelope primitive. In general, an i_process should not attempt to allocate resources during execution since there is a real probability that there will not be any resources available. Also, any kernel primitive which can potentially perform a process switch should check to ensure that it does not attempt to perform a process switch on an i_process.

The i_process could be considered as a high level "intelligent" handler of an interrupt. Thus, it might be desirable for the i_process to maintain state information between interrupts (executions). The design of each i_process should take this into account. Some issues include: is it required to perform a complete context switch to run an i_process or can it be invoked as a simple C-function (in either case a PCB would be required); how would state information (memory) be maintained across invocations; what additional private data structures would be needed by each i_process, etc..

An i_process has some privileges that a user process does not. For example, it can call directly the kernel version of the user visible API without call calling the user visible API functions and entering the kernel by a SWI. It is already in the kernel due to an interrupt handling sequence.

8.1.1 Example

As an example, consider the case where a user process sends a message to a UART-i_process. The message text contains a string of characters to be send to the UART. The i_process delivers each character to the UART one-at-a-time on each TX interrupt. On each TX interrupt from the UART, the interrupt handler starts the respective i_process. The i_process does a receive to see if there are any new messages sent to it, and if yes, places them on some private service queue. It then checks its previous state to see which character (if any) it must send to the UART from the current message envelope on its service queue. If the last character has been sent from the current message envelope, the message envelope is sent back to the requesting process as an acknowledgement. Next, the service queue would be checked to see if more messages are waiting for service. When the last character of the last message is sent, then a state variable is set to indicate that the i_process is idle and that the UART is in a ready state since no further TX interrupts would be caused by the UART until it is loaded with another character.

8.1.2 Possible Complications

The previous section described an i_process sending characters to a UART in response to TX interrupts and an overview of its functionality. Generally, a UART will only give one interrupt when its transmit buffer is empty. Usually, this is sufficient for the i_process to load the next character to the UART. What happens when the i_process has sent the last character in the last message on its service queue and it has returned the message envelope. Potentially, the i_process will never run again since the TX interrupt handler will not run because there will be not any more TX interrupts until the transmit buffer is loaded again. So, if messages are sent to the i_process, the i_process will never receive them and the characters will not be sent to the UART since the i_process must be started by the interrupt handler which will not run. So, we have a problem. The same situation occurs after initialization and the first message is sent to the i_process. A similar scenario may be possible for other i_processes defined in the system. This type of problem does not occur for periodically occurring interrupts and their associated i_processes. What we need is to either avoid this situation (?) or perhaps have a way to "jump-start" an i_process for the first time (or periodically). The possible solutions are left as an exercise.

8.2 A generic interrupt handler

We can now update the previous generic exception handling sequence to, assuming non preemptive for now, with the following pseudocode:

exception_handler:

```
begin
        // code here to save current_process context into its
        //context save area as defined by its PCB
        save_PCB = current_process
        select interrupt source
                A:      current_process ← i_proc_A_pcb
                        //restore i_proc_A context
                        //invoke i_proc_A handler
                        break
                        ........

                Z:      current_process ← i_proc_Z_pcb
                                        //restore i_proc_Z context
                                        //invoke i_proc_Z handler
                                        break
                        end select

                //code to save context of interrupt handler (i_process)
                        current_process = save_PCB;

                //code to restore current_process context
                //
                //perform a return from exception sequence
                //this restarts the original process before i_handler
                end;
```

For a preemptive system, a check would be made on the return from the i_process to determine if the ready queue contains a ready process of higher priority than the process that was executing at the interrupt event. If yes, then the original process would be preempted by the process which was made ready by the i_process causing an unscheduled process switch.

There is another design issue to consider. During its execution, an i_process may need to invoke a kernel primitive. Does the i_process use the same user API interface that a user process uses, or does it directly call the kernel primitive? Recall that the i_process executes during interrupt handling within kernel mode. Thus it could bypass the user visible API interface and directly call the kernel version of the primitive avoiding any overhead and complexity of re-entering the kernel with a TRAP instruction. Since kernel primitives will not block an i_process, there is no danger of the i_process becoming blocked within the kernel and requiring a process/context switch to be performed.

8.3 Timing Services

This is one of the more fundamental required services. Real-time processes interact with their environment and need services such as:

- delay : delay my execution for *n* seconds. Voluntarily give up execution until the specified time expires, then put back on ready queue (often referred to as sleep).
- timeout : requests kernel to inform process when a specified time period has expired, the process continues execution.
- repetitive timeout : request to be informed every *n* microseconds until cancelled

These requests could be in:

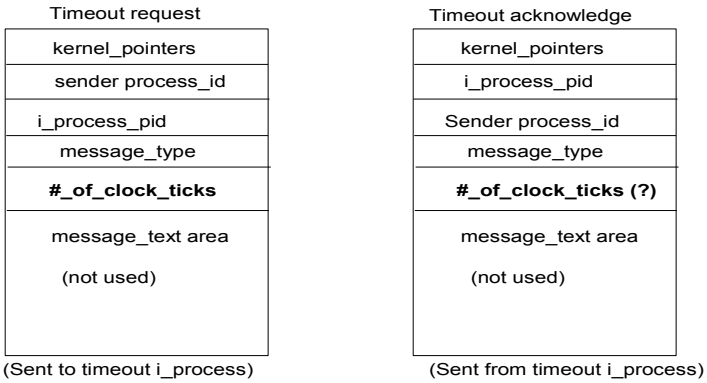
- relative time : *n* number of clock ticks from now
- absolute time : for example, February 24, 10:34:22 AM, 2008

as appropriate. "*clock ticks*" are defined as the number of interrupts from a hardware device which periodically sends an interrupt to the processor. For example, the processor could receive an interrupt every 10 ms. In that case a "clock tick" would be 10 ms in duration.

The basic design issues include: what is the protocol to access the timing services; "who" performs the services; and how is the invoking process informed of the timing service's results? Since we assumed a message passing RTX, we can construct the timing services within that framework. We already have the concept of an i_process. Let a user process send a message to a special i_process which performs the service. The i_process then sends a message back to the requesting user process as an acknowledgement.

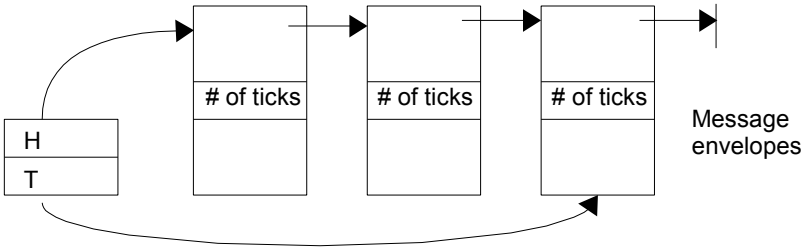
Since message passing is the basic form of IPC, we can add an additional field to the message envelope structure when it is used to communicate with a timing service i_process. Consider a simple timeout service accessed by the standard `send(target_pid, message_ptr)` primitive. The adjacent figure depicts the resulting message envelope with a new field called "# of clock ticks".

All user processes 'know' the pid of the timeout i_process. After the expiration of the time, the timeout i_process sends the original message envelope back to requester. The timeout service maintains requests in a private ordered queue. There are many possibilities in the type of queue and the insertion of timing requests into the queue. A design goal would be to minimize the overheads involved for i) the insertion of messages onto the queue and ii) to determine that the timeout interval has expired for any individual request.



At each clock tick (hardware timer interrupt), the timeout i-process executes. The i-process invokes the receive message primitive to check for any new requests since the last clock tick interrupt. Since it is an i_process, it cannot block if there are no messages waiting for it. If there are any new requests, then they are added to the sorted list. Then the i-process checks if any timing requests have expired expired. If yes, then remove the message(s) from the sorted timeout list and then send the message envelopes back to requesters as an acknowledgement of the timeout expiration.

There are many possibilities of how to maintain a sorted timeout list. As an example, consider the figure below



Timeout queue

which depicts one such sorted list. The # of ticks field can be modified by timeout to indicate the # of ticks after its predecessor in the list. New timeout requests would be inserted in sorted relative timeout order. As an example consider a timeout queue with the following # of ticks values:

- queue contents of timeout list = {25, 30, 0, 10} (the # of ticks field shown)
  - one timeout for 25 clock ticks
  - two timeouts for 55 clock ticks
  - one timeout for 65 clock ticks

Each request must be inserted into the appropriate location of the queue and all requests which follow it in the queue might also have their clock tick adjusted. After insertion, checking for a timeout expiry requires only to decrement the request at the head of the queue (see the following pseudocode). This method requires some overhead to install the timeout request onto the timeout queue (done once per request) but can very efficiently determine when a timeout expires.

In addition, the timeout i_process can also maintain the internal kernel clock. This clock is based on clock ticks and is simply an integer (i.e. 32-bit) counter which is simply incremented at each clock tick interrupt. This clock can also be used to perform the timeout operation. For example, for each timeout request received, add the specified number of clock ticks to the current kernel clock and store this value as the timeout expiry time in the received message envelope. The timeout queue would be sorted using this field. Then, on each interrupt simply compare the expiry time of the head of the timeout queue with the incremented kernel clock. A timeout has expired when the timeout request is less than or equal to the kernel clock.

The following is a basic outline of service execution for an interrupt based on maintaining timing requests in a relative sorted order as discussed previously:

```
timeout_i_process:
{
        env ← K_receive(); //invoke primitive to get pending msgs
        while (env is not null)
        {
                //code to sort the envelope into the timeout_list
                env ←K_receive(); //see if any more msgs left
        }
        if (timeout_list is not empty)
        {
                //code to decrement the tick count of 1st in the timeout_list
                while (head_of_timeout_list[tick_field] is zero)
                {
                        env ← dequeue( timeout_list );
                        temp ← env[source_pid];
                        env[source_pid] ← timeout_i_process_pid;
                        K_send( env, temp );    //return envelope
                }
        }
```



```
}
```

9 Process Preemption

For the system described, preemption is simple to add. For example, each time a process is placed on a ready queue by some primitive (e.g. possibly by a send, deallocate memory or message envelope, or by the operation of an i_process) perform the following:

- check if the ready queue contains a ready process whose priority is higher than the currently executing process,
- if yes, then perform a process switch (at appropriate time) to the higher priority process.

A check would also be made by the interrupt handler just before control is returned to the process executing at the occurrence of the interrupt. Thus a process switch could be made just before the RTE of the interrupt handler.

Key design issues are "at what point is the process context saved?" and "at what point is a process switch made?". For example, when a user process invokes a kernel primitive, is the actual process context saved by the SWI handler before the kernel primitive is called by the SWI handler? Or is the complete process context saved by the context switch function if there is a process switch. Which stack pointer is used after entry into the kernel by a TRAP instruction: the user's stack pointer or do we switch to a special kernel stack pointer? These are all design issues and left to you to explore.

10 Process Switching – Context Switching

Performing a process switch requires a more detailed examination. A process switch can be performed at the TRAP handler boundary (after returning from a kernel primitive) or within a kernel primitive itself. In this example, we allow a process switch to occur within a kernel primitive as needed. Recall that a process switch will be requested either by a kernel primitive while processing a user request or by the interrupt handler when a process preemption is required due to the action of an i_process. Lets examine the case of a process switch requested by a kernel primitive and as an example consider the receive message primitive which is repeated below:

```
K_receive() {       //need a check for i_processes

   if ( current_process's msg_queue is empty) {
         set state of current_process to blocked_on_receive
         process_switch( );
        //*** return here when this process executes again
   }
   env ← dequeued envelope from the process' message queue
   return  env

}
```

In this example, the current process will block inside the kernel and will eventually restart execution within the kernel as well. After the process restarts, it will return to the next line of code following the call to process_switch(). The net effect is that of a simple call to some function called process_switch() which eventually returned and the kernel primitive continues execution without any knowledge that the process was blocked for some time interval. Hence, all the "complexity" is within the function process_switch().

Process_switch is not informed of the process requesting a process switch since the kernel variable `current_process` is pointing to the PCB of the process that invoked receive in the first place. What functionality

does process switch have? At a minimum, in this example, it will call `rpq_dequeue()` to obtain a pointer to the PCB of the highest priority ready process (the process which will execute next). Thus the simplest basic skeleton of process switch could be:

```
process_switch( )
{ PCB_ptr *next_pcb;      //local variable to hold ptr to next pcb
  next_pcb = rpq_dequeue( );    //get ptr to highest priority ready process
  context_switch( current_process, next_pcb );
 }
```

Process_switch does not enqueue the `current_process` 's PCB on any queue nor modify the status field – this must be done by the caller of process_switch. Process_switch could change the status field of next's PCB to executing. Thus, the current process will be "stopped" inside of "context_switch()". The next process will restart inside of context_switch(), then return to process_switch and then return to whoever called process_switch (assuming that the next process was also stopped in this manner). Fundamentally, when a process restarts inside of context_switch, it will return to what ever function called context_switch. Care should be done in context_switch's design to made it "bullet-proof" and allow it to be called by any function (not just process_switch).

The next issue is: what is the design of "context_switch"? At this point, context_switch has some fundamental requirements:

1. Save the context of the current process into its PCB save area allowing it to restart such that a return will be made to the caller of context_switch. This includes, the stack pointer, PC and SR.
2. Restore the context of the next process (from its PCB save area including the SP, PC and SR)).
3. Possibly set the status field of the next's PCB to executing.
4. Possibly set the kernel variable `current_process` to point to next's PCB.
5. Restart the next process such that it "returns" to who ever called context_switch.

The structure of context_switch is:

```
context_switch( pcb_ptr *current, pcb_ptr  *next) {
    char * temp1;  //some temporary variables
    char * temp2;
    // code to save context of 'current' process
    // possibly set next->status = executing
    // possibly set "current_process" to point to next's PCB.
    // code to restore context of 'next' process
    ASM("RTE");              // return from exception
Comehere: ASM("nop");
          ASM("nop");
          return;    //normal C-language return statement
}
```

What is the value of "PC" that will be saved in the PCB of "current"? When 'current' restarts, we want it to return to the caller of context_switch. This is easy, we actually want it to restart at the label "`Comehere:`". Therefore, when the current process eventually restarts, it will execute the "`nop`" instructions and then hit the return of `context_switch` and thus returning to `process_switch` (or whoever called it) using the normal C-language conventions. So we need to add code to get this value (the address of `Comehere:` label) and save it into current process's PCB field for PC. The RTE instruction requires a specific sequence of data on the stack (pointed to by a7) prior to the execution of the RTE. Look in the MCF5307 programmer's manual for any details. Basically, the following sequence is performed:

$2 + (SP) \rightarrow SR;\quad 4 + (SP) \rightarrow PC;\quad SP + 8 \rightarrow SP$

The value of PC and SR are obtained from next's PCB save area and are pushed onto the stack last in the proper

order. In addition, the Format/Vector fields must contain an appropriate bit sequence. You can push these onto the stack with the SR value. Unless you want to become a “guru” MCF5307 assembly language programmer, avoid pushing any 8 or 16 bit values onto the stack in your program. Always maintain a long-word aligned stack pointer.

From the above, we observe that if a process switch was made by the `process_switch` function, then the process will always restart within the `context_switch` function. If the process was 'stopped' by an interrupt hander (due to preemption), then the process switch operation will simply restart the next process to its logical restart point, not necessarily within the `context_switch` function.

With some thought, we might be able to use the basic ideas here to perform a process preemption handled by an interrupt handler after an i_process makes a high priority process ready; perhaps even using a 'bullet-proof' version of the context_switch just discussed.

11 A zero security minded RTX?

One of the assumptions for the kernel design was based on user processes that were friendly, cooperative and always did the 'right thing' and were non-malicious. This assumption 'might' be valid in the final system but may be optimistic during the initial stages during the testing and debug phases of user processes. In the current design, it is rather easy to crash the kernel due to a bug in a user process or by a malicious action on the part of a user process. Perhaps the first line of defence would be the hardware design of the embedded microprocessor system itself. The hardware should make use of processor generated output signals indicating user mode and supervisory mode execution to ensure that no kernel address space is accessed by a user process while the processor is in user mode execution. Such attempted access should generate an exception condition forcing the processor to perform exception handling with kernel supplied exception handlers. Also, if available, virtual memory can play a valuable role. The virtual to physical memory page table translations can ensure a separation of address spaces between all user processes and the kernel address space preventing any unwanted interaction. Further discussion on these points is beyond the intended scope of this document.

The current design can be upgraded to provide a minimal level of security during initial system debug (with minimal performance impact) and then be disabled (??) when the system proves to be stable. First, consider the primitives `send`, `receive` and `deallocate_envelope` . For `receive`, the user process assumes that the kernel has provided a valid pointer to a message envelope. For `send` and `deallocate_envelope`, the kernel trusts that the user process is providing a valid pointer to some kernel generated (during initialization) envelope (block of memory). The kernel does no check to verify the validity of the supplied envelope pointer. If this pointer is invalid, then a crash of the system may be inevitable. Also, the kernel does not track ownership of message envelopes after they are allocated. There is nothing to prevent a user process from deallocating an envelope after it has sent the envelope to another process which could result in a system instability. A process could write into the envelope a message text which overflows the length of the envelope and possibly overwrite another envelope. Ideally, these problems 'should' not arise in the final production system, but could occur during the debug phase.

The kernel can easily place all envelopes (created during initialization) on a linked list using the kernel pointers field within each message envelope and then scan the list to verify a valid pointer is passed to the kernel. But a user process most likely also has access to these fields and could easily change them causing instability within the kernel since theses fields are also used to place free envelopes on the envelope free list. This simple approach adds some level of security but can be defeated by an errant process.

Assume there is at least some separation of kernel and user address spaces provided by the hardware design of the controller. This would prevent wholesale invasion of a user process into the kernel's private data structures. There

are many possible approaches to verify that a valid envelope pointer (which a process currently owns) is provided by a user process during some primitive invocation. The following describes a simple approach that can provide a level of security. This scheme can easily be optimized and left as an exercise.

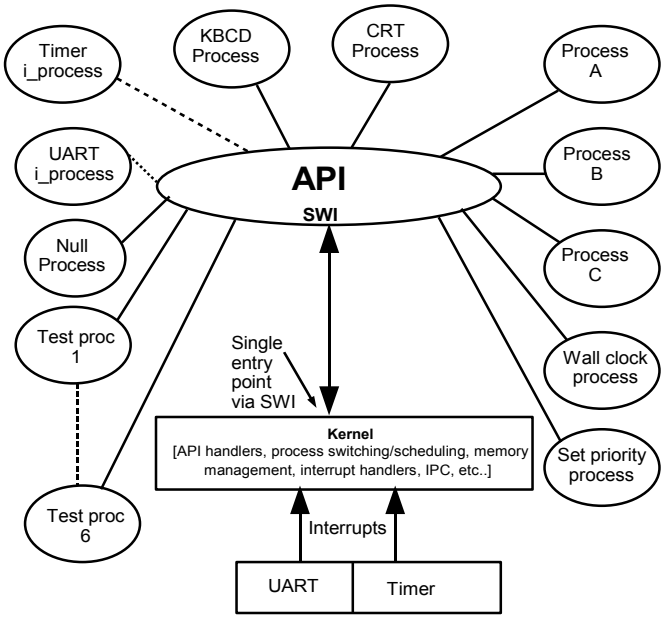
We can define an envelope control block (ECB) similar in structure but simpler than a PCB. During system initialization, one ECB is created for each envelope memory block. It is the ECB, not the message envelope, that is placed on the envelope free queue. The message envelope's structure is still the same with all fields except it is missing the kernel pointers field since it is not needed. The kernel also maintains a linked list of all ECBs in the system. The ECB contains (at a minimum) a kernel pointers field (used to place it on various linked lists), a pointer to the actual memory block of the envelope the ECB represents and a field containing the PID of the current owner of the envelope.

On a message send, the send primitive would copy the destination process's PID into the current owner field of the ECB. The allocate envelope primitive would copy the requesting process's PID into the current owner field of the ECB when an envelope is allocated; and obtain the pointer to the envelope from the appropriate field within the ECB. The deallocate envelope primitive would reset the current owner field of the ECB indicating that the kernel is the owner. Thus any attempt to send or deallocate a message envelope not owned by the requesting process can be easily detected by the kernel's primitives. Also, any attempt to deallocate/send an envelope by supplying an illegal envelope pointer can be easily detected. Depending on the implementation, detecting an illegal envelope pointer is perhaps the largest overhead without a consistent execution time. The overheads of maintaining the current owner field in the ECB is negligible.

Other simple extensions (causing minor revisions to existing primitives) can include the verification that the destination process PID in a message send is valid, etc.

12 Appendix A: User API

The following figure depicts a simplified view of the overall process-kernel structure of the project. You might have additional processes in your design. Processes access RTX services via a single SWI (Software Interrupt) from the specified user visible API. The i_processes are shown with dotted lines since they run in kernel mode so



they might access the kernel version of the API functions directly rather than following the user process protocol. In a typical system, user processes and user visible API functions can not access the protected address space of the kernel. None of the kernel functions or data structures are visible or accessible while executing in user address space. These are only available in the kernel's address space while the processor is executing in "supervisor mode". For example, the user visible/called API "receive_message" can not itself perform the requested service. Basically, it passes the request to the kernel by executing a software interrupt instruction (a SWI or TRAP instruction on most processors). This causes the processor to begin an exception handling sequence which starts the kernel's trap handler (in the kernel address space and in the processor's supervisor mode of execution).

12.1 Example of a single kernel entry point design.

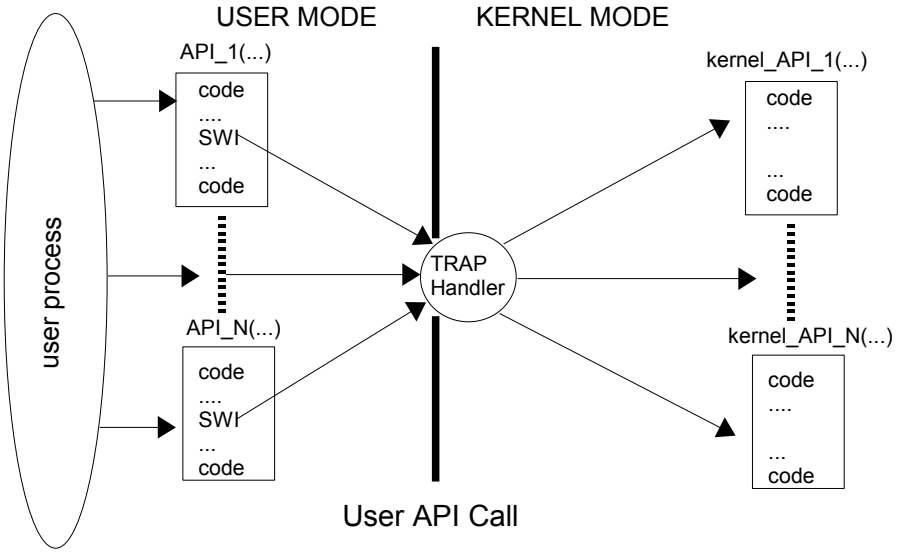
All entries to kernel are by the same SWI (a TRAP in the MCF5307) instruction in the user process visible API_i(..) function. All API_i(..) functions invoke the same SWI instruction after presetting various CPU registers to indicate to the Kernel_API_i(...) of the functionality required plus any additional parameters obtained from the stack frame of the invoking process.
The SWI trap handler (within the kernel) would examine the CPU registers and determine which Kernel_API_i(..) function to invoke. The trap handler would save some/all of the process context (depending on the design) and ensure that interrupts are disabled during the kernel's execution. The trap handler would also disable some (or all) interrupts ensuring atomicity while in the kernel.

Comments:

In general, the API functions (as described in the project document) should **not** access the private variables, data structures, queues, etc. of the kernel. Only the respective Kernel_API_i(..) functions have the privilege of doing so. The specified behaviour is performed by the Kernel_API_i(..) function not by the published API, since these only serve to provide a controlled entry point into the kernel.

The user visible API functions would perform (at least) the following:

1. for an API function call with *i* formal parameters, move registers d1 .. d*i* onto the stack (save them).
2. Copy the values of the actual parameters (from current stack frame) into registers d1 .. d*i*.
3. Load register d0 with an appropriate Kernel_API_*k* integer code (used to select which function to invoke).
4. Perform a software interrupt (SWI) using an appropriate TRAP #*n* instruction where *n* is the trap number.
5. The kernel's trap handler would determine which kernel_API function to invoke and pass to it the required parameters obtained from d1 .. d*i* registers; and then finally return to the user API by a return-from-exception instruction.
6. On return from the TRAP, register d0 should already be pre-set with the required return value to be returned to the user process.
7. Now, restore registers d1 .. d*i* previously saved on the current stack frame.
8. Use the gcc supplied return mechanism to return control back to the original caller.



An example from the course project:

As an example of where this information can be used, consider the project's user process visible send message API function:

`int send_message (int process_ID, char * MessageEnvelope)`

In our implementation, `send_message` is just a skeleton API function (a stub) with no real functionality except to pass the request to the kernel, which will then perform the send message functionality, and then return the results to the process which invoked `send_message`. The user visible `send_message` does not have the right (or ability) to disable interrupts nor access internal kernel data structures. The request is transferred to the kernel via a trap instruction. The kernel's trap handler examines the request code placed in register `d0` (after disabling interrupts and ....) and calls the respective kernel function supplying the parameters available in the `d1` and `d2` registers. In this example, the actual functionality of `send_message` would be in a C-function called:

`int kernel_send_message (int process_ID, char * MessageEnvelope)`

The trap handler would call `kernel_send_message`. Now, `kernel_send_message` runs in kernel mode and can access all kernel variables, queues, etc without any interference from interrupts and user processes.

Another item for consideration is: would an i_process call `send_message` or `kernel_send_message`? Recall that i_processes only run during interrupt handling, so.....

Below is an example (of many possible) implementation of the `send_message` API stub.

```
int send_message( int process_ID, char * MessageEnvelope)
{
   int retCode = 0;
   asm("move.l %d1, -(%a7)"); //save the d1 register on stack
   asm("move.l %d2, -(%a7)"); //save the d2 register on stack
   asm("move.l 8(%a6),%d1");  //copy 'process_ID' into d1 register
   asm("move.l 12(%a6), %d2");       // copy 'MessageEnvelope' into d2 register
   asm("move.l #7, %d0");            //assume that send_message has kernel_code of 7
   asm("TRAP #0");                   //assume trap#0 is the kernel swi trap handler
                                     //we assume that the return value will be in reg. d0
   asm("move.l (%a7)+, %d2");   // restore reg. d2
   asm("move.l (%a7)+, %d1"); // restore reg. d1
   asm("move.l %d0, -4(%a6)");   //copy returned value to local variable 'retCode'
   return retCode;
}
```

Recall the MCF5307 trap instruction sequence:

```
1 → S-Bit of SR
SP – 4 → SP; nextPC → (SP); SP – 2 → SP;
SR → (SP); SP – 2 → SP; Format/Offset → (SP);
(VBR + 0x80 + 4*n) → PC
where n is the TRAP vector number
```

The trap handler would do an initial save of registers (some or all, depending on design) and then examine the d0 register to determine which kernel primitive is to be invoked. This could be done by a C-language switch statement after saving d0 into some local variable. Before calling the appropriate C-language kernel primitive, push the required arguments onto the stack (using assembly language statements) – these were in the d1, d2, d3 registers when the TRAP was executed and then call the kernel primitive using a "bsr" assembly language call (or place the contents of d1, d2, d3 into local variables and then using C-language call the kernel primitive with appropriate actual parameters). On return, the kernel primitives' return value will be in the d0 register (gcc default). Next re-align the stack by adding an offset to account for the passed parameters (still on the stack) – if you called the kernel primitive with a "bsr". Restore any previously saved registers (be sure to leave the new value of d0 in d0) and execute a RTE instruction which should return to the original user level API function that executed the TRAP instruction. See the section on Process/Context switching for details of a process switch occurring when a kernel primitive is called.

13 Appendix B: Using C with Assembly Language

During the course of the project, you will need to access local/global variables and manipulate the run-time stack structure with assembly language, often within a C-language function. You need to know how and the order in which actual parameters are stacked for a function call as well as how/where the local function variables are accessed and where they can be found.

This short tutorial gives some insight into how to access local function variables and how to pass and access actual parameters of a C-language function call using gcc and the MCF5307 processor. You should already be familiar with assembly language and the MCF5307 processor from earlier courses.

Consider the following simple set of 2 C functions used to illustrate how gcc passes parameters, where local variables are stored and how to access local variables and passed parameter values:

```
int simple1( int a ,int b)
{
    a = b;
    return 0;
}

int simple2( int x, int y)
{
  int c;
  int d;
  c = x;
  d = y;
  d = simple1(c,y);
  c = d;
  return d;
}
```

We compile this source file on eceunix using the following command line which just generates the assembly file called simple.s for the source file simple.c:

```
m68k-elf-gcc -Wall -nostdlib -m5200 -Tmcf5307.ld -S simple.c
```

The results of the compile, the file simple.s, is shown below:


```
        .file   "simple.c"
gcc2_compiled.:
.text
.globl simple1
        .type    simple1,@function
simple1:
        link.w %a6,#0
        move.l 12(%a6),%d0
        move.l %d0,8(%a6)
        clr.l %d0
        jbra .L2
.L2:
        unlk %a6
        rts
.Lfe1:
        .size    simple1,.Lfe1-simple1
.globl simple2
        .type    simple2,@function
simple2:
        link.w %a6,#-8                  //line A
        move.l 8(%a6),%d0
        move.l %d0,-4(%a6)
        move.l 12(%a6),%d0
        move.l %d0,-8(%a6)
        move.l 12(%a6),-(%sp)
        move.l -4(%a6),-(%sp)
        jsr simple1
        addq.l #8,%sp                   //Line C
        move.l %d0,-8(%a6)
        move.l -8(%a6),%d0
        move.l %d0,-4(%a6)
        move.l -8(%a6),%d1
        move.l %d1,%d0
        jbra .L3
.L3:
        unlk %a6                        //Line B
        rts
.Lfe2:
        .size    simple2,.Lfe2-simple2
        .ident  "GCC: (GNU) 2.95.3 20010315 (release)(ColdFire patches - 20010318 from http://fiddes.net/
coldfire/)"
```

Recall the material from E&CE 251 and the run-time environment produced by compilers.

First, for the MCF5307 processor, gcc uses register a6 as a frame pointer.

When gcc generates a function call with actual parameters, then gcc will push the values of the parameters onto the stack in the reverse order. For example, if a call is made to "`simple1(c,y)`", then gcc will push the value of "`y`" onto the stack first, followed by the value of "`c`" and then make the call. Also, the `d0` register is used to return a value to the caller.

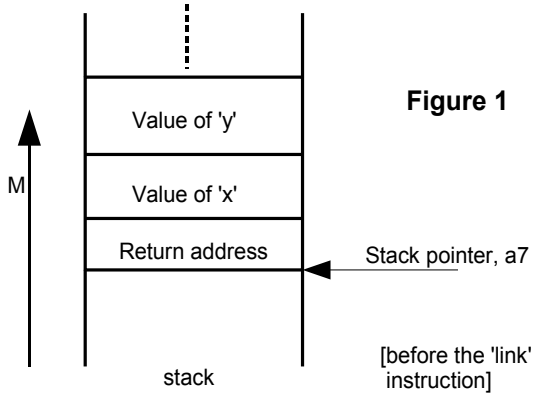


Figure 1

When execution arrives at the call site, then (i) the caller's frame pointer (register a6) must be saved, (ii) a new frame pointer is placed in a6, and (iii) storage on the stack must be allocated for the local variables of the called function, and (iv) a new value for the stack pointer (register a7) must be generated. All of this activity is performed with a single instruction, the `link`, see **line A** above. As an example, lets assume that the function `simple2(int x, int y)` has been called and the values of the actual parameters have been stacked. Before the link instruction at **line A** is executed, our stack looks as shown in Figure 1. After the link instruction (**line A** above) is executed, the state of the stack frame is shown in Figure 2. Now, we can use simple2's frame pointer (a6) to access the passed values and the local variables by simply specifying an offset from a6. For example (using Figure 2):

- the value of 'x' can be moved into register `d0` with "`move.l 8(%a6),%d0`"

- and variable 'd' can be set with the contents of register `d0` with "`move.l %d0,-8(%a6)`"

Just before function simple2 returns, see **line B**, we execute the " `unlk %a6`" instruction which restores the stack and the caller's frame pointer to that shown in Figure 1. This also deallocates the space occupied by the local variables. Then the " `rts` " instruction finally returns to the caller who restores his stack to the before-call state (e.g: see **Line C**).

At this point it should be obvious of what must be done to call a C-function from assembly language and how to

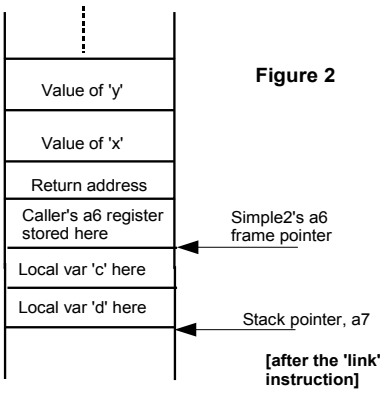


Figure 2

access various variables using assembly language within a C-language function.

13.1 An example from the course project:

As an example of where this information can be used, consider the project's user process visible send message API function:

`int send_message (int process_ID, char * MessageEnvelope)`

In our implementation, `send_message` is just a skeleton API function (a stub) with no real functionality except to pass the request to the kernel, which will then perform the send message functionality, and then return the results to the process which invoked `send_message`. The user visible `send_message` does not have the right (or ability) to disable interrupts nor access internal kernel data structures. The request is transferred to the kernel via a trap instruction. The kernel's trap handler examines the request code placed in register `d0` (after disabling interrupts and ....) and calls the respective kernel function supplying the parameters available in the `d1` and `d2` registers. In this example, the actual functionality of `send_message` would be in a C-function called:

`int kernel_send_message (int process_ID, char * MessageEnvelope)`

The trap handler would call `kernel_send_message`. Now, `kernel_send_message` runs in kernel mode and can access all kernel variables, queues, etc without any interference from interrupts and user processes.

An other item for consideration is: would an i_process call `send_message` or `kernel_send_message`? Recall that i_processes only run during interrupt handling, so.....

Below is an example (of many possible) implementation of the `send_message` API stub.

```
int send_message( int process_ID, char * MessageEnvelope)
{
   int retCode = 0;
   asm("move.l %d1, -(%a7)"); //save the d1 register on stack
   asm("move.l %d2, -(%a7)"); //save the d2 register on stack
   asm("move.l 8(%a6),%d1");  //copy 'process_ID' into d1 register
   asm("move.l 12(%a6), %d2");       // copy 'MessageEnvelope' into d2 register
   asm("move.l #7, %d0");            //assume that send_message has kernel_code of 7
   asm("TRAP #6");                   //assume trap#6 is the kernel swi trap handler
                                     //we assume that the return value will be in reg. d0
   asm("move.l (%a7)+, %d2");   // restore reg. d2
   asm("move.l (%a7)+, %d1"); // restore reg. d1
   asm("move.l %d0, -4(%a6)");   //copy returned value to local variable 'retCode'
   return retCode;
}
```

After compiling with the '-S' option we get the following:

```
        .file   "send_mess.c"
```



```
gcc2_compiled.:
.text
.globl send_message
        .type    send_message,@function
send_message:
        link.w %a6,#-4
        clr.l -4(%a6)
#APP
        move.l %d1, -(%a7)
        move.l %d2, -(%a7)
        move.l 8(%a6),%d1
        move.l 12(%a6), %d2
        move.l #7, %d0
        TRAP #6
        move.l (%a7)+, %d2
        move.l (%a7)+, %d1
        move.l %d0, -4(%a6)
#NO_APP
        move.l -4(%a6),%d1
        move.l %d1,%d0
        jbra .L2
.L2:
        unlk %a6
        rts
.Lfe1:
        .size    send_message,.Lfe1-send_message
        .ident "GCC: (GNU) 2.95.3 20010315 (release)(ColdFire patches - 20010318 from http://fiddes.net/
coldfire/)"
```

14 Index

To be done.

