Introduction

Presto is a pre-emptive priority-driven real time operating system with support for timers, interprocess communcation, semaphores and dynamic memory.

A brief history

Early in my programming career, I was fascinated by multi-tasking systems. I got a special thrill the first time I wrote an interrupt service routine. Here was a unique piece of code that was executed outside of the main thread of my host program.

This fascination quickly turned into a necessity when I was asked to add a network interface to an existing product. What evolved out of that project was a cooperative multi-tasking scheduler. Simply put, I kept a list of functions that I needed to call, and I called them until they were all gone. Except each time I called one function, new ones might be added to the list. This simple and elegant solution solved that particular problem, and that system is still in production today. But a deeper problem bothered me.

How could I pre-emptively swap between multiple threads? What kind of magic is involved in *stealing* the processor away from a task, and giving it to another task?

This question is at the heart of Presto, and answering it has been the primary inspiration behind the development of Presto.

As luck would have it, several things happened at nearly the same time, and so the secrets of multi-tasking were revealed. I decided to build a simple single-board computer that would be me development platform. My friend Greg Preston convinced me to build a Motorola 68HC11-based "Handyboard". Presto takes its name from Greg's last name (plus it has a magical sound to it). At the same time, I read an article in *Circuit Cellar* magazine about a simple time-slicing system that ran on the HC11. I studied the example, and how they carefully set the stack up and then generated an interrupt which changed the stack pointer. I now had enough knowledge to build my RTOS.

After a weekend of tinkering, I had my first prototype kernel swapping tasks. It worked fairly well, but I had a small bug that caused the system to hang after a few seconds of operation. Little did I know, it would be four years before I found and fixed that bug.

I studied the code, and I ran it on an emulator program, and I took the board to work and run it on a logic analyzer, but the bug still eluded me. Since I was working full time and going to graduate school, I tended to put my project on a shelf for months at a time. When I did have time to devote to it, it was often only a few hours, and I spent most of that time getting re-aquainted with where I had left off. I graduated. I moved to Singapore. I had a baby. Where was I going to find some free time?

I decided that I needed to inspect every instruction that ran on the processor. I stay up late after the baby went to sleep, surfing the net for every emulator I could find. I decided that they fell into two categories: those that emulated the processor correctly but had horrible user interfaces, and those that were easy to use but did not emulate the processor correctly. Specifically, I needed to be able to run a million instructions without pressing a button a million times. And I needed to generate timer interrupts.

I finally gave up on the net, and wrote my own emulator. It was tedious work, but it was fun. I compared the log files that I generated to the logs of the other emulators to find potential bugs. Wherever they differed, I consulted the processor manual to see which one of us was correct.

Still the bug eluded me. I moved back to the US. We were about to have another baby. I knew that I could kiss those precious moments of free time away. So with time ticking away, I focused on the problem by running the simulator for hours and generating huge log files. I wrote a perl script that would read the log files and summarize what was going on. What function was the program counter pointing to? Where was the stack pointer pointing? Was it within one of the task stacks?

Eventually, my perl script determined that my stack pointer had drifted out of one task's stack. In fact, it had popped one too many bytes off of one task's stack. Depending on what was in that doomed location of memory, the result might have been catastrophic, or it might have gone unnoticed. Searching the code, I quickly found that in one particular instance, the stack would get clobbered. It only occured when the kernel decided to re-evaluate which task to run, and the winning task was the same one that was already running. I solved the problem that night, and it felt great.

The good news is that in the process of trying to find a bug over a period of four years (sitting the project on a shelf for months at a time and dusting it off again for a late night of bug-hunting), I wrote and re-wrote the entire kernel a couple of times. So it was an evolution of small improvements.

Presto today

Presto has been structured with elegance and simplicity as primary goals. It is intended to be small and modular, and easily ported to new processor architectures. It is written almost entirely in C, with a few sprinkles of inline assembly language (only where necessary, in the heart of the kernel).

I borrowed concepts that I had seen elsewhere, things that were pleasing in their simplicity, and incorporated them into Presto. Just before version 1.0, I decided to change the basic metaphor that Presto used to schedule tasks, and in the process I divided the code into the sections that you see today: the core kernel, the task-switching peripherals (mail, timers, sempahores) and dynamic memory management.

Presto is not the Cadillac of real time operating systems. Instead, it is more of a hand-crafted go-cart.

How Presto is organized

Tasks and priorities

The kernel

Task-switching perpherals: Mail, Timers and Semaphores

Dynamic memory

The kernel

This module is the heart of Presto. Here is where we keep our list of tasks, and where we evaluate priorities to pick who runs.

In order to do pre-emptive multitasking, we have to perform context switches. This is a pretty simple operation, but it can be tricky to get right. Basically, all of the registers are saved to the stack and then the stack pointer is saved in the task's TCB. The new task's stack pointer is read from its TCB, and then the registers are pulled from that task's stack (remember, each task has its own stack).

The process of determining who should run next and then doing a context switch has been cleanly encapsulated in an interrupt service routine. That way, whenever some event happens that makes a new task ready, we simply issue a SWI (software interrupt).

In order to tell if a task is ready to run or not, Presto uses "ready bits" which I call "triggers". Each task has eight triggers (this can be changed in configure.h if needed). If a task is waiting, then its wait_flag will be set to something other than zero. When one or more of the triggers that it is waiting for becomes set, then that task will become ready.

The tasks are kept in a linked list (in priority order), so it is easy to traverse the list to find the highest priority ready task. Presto supports temporary over-riding of priorities (in which case the task will be moved to a different place in the linked list). This feature is used by the "priority inheritance" feature of semaphores.

Kernel data types

// core kernel

```
typedef KERNEL_TASKID_T       PRESTO_TASKID_T;

typedef KERNEL_TRIGGER_T      PRESTO_TRIGGER_T;

typedef KERNEL_PRIORITY_T     PRESTO_PRIORITY_T;
```

// time

```
typedef KERNEL_INTERVAL_T     PRESTO_INTERVAL_T;
```

Start-up

```
extern void presto_init(void);

extern PRESTO_TASKID_T presto_task_create(void (*func)(void), BYTE * stack, short

extern void presto_scheduler_start(void);

extern void presto_wait_for_idle(void);
```

Task Priorities

```
extern PRESTO_PRIORITY_T presto_priority_get(PRESTO_TASKID_T tid);

extern void presto_priority_set(PRESTO_TASKID_T tid, PRESTO_PRIORITY_T new_priori

extern void presto_priority_override(PRESTO_TASKID_T tid, PRESTO_PRIORITY_T new_p

extern void presto_priority_restore(PRESTO_TASKID_T tid);
```

Triggers

```
extern PRESTO_TRIGGER_T presto_wait(PRESTO_TRIGGER_T triggers);

extern void presto_trigger_set(PRESTO_TRIGGER_T trigger);

extern void presto_trigger_clear(PRESTO_TRIGGER_T trigger);

extern void presto_trigger_send(PRESTO_TASKID_T tid, PRESTO_TRIGGER_T trigger);

extern PRESTO_TRIGGER_T presto_trigger_poll(PRESTO_TRIGGER_T test);
```

Mail

This module implements a simple and fast mechanism for passing messages between tasks. The metaphor used is that of mailboxes and envelopes.

A task can own one or more mailboxes. Each mailbox has associated with it, a "trigger" (or "ready bit"). When mail arrives in the mailbox, the trigger is set. If the task is waiting on that trigger, then the task will become ready.

There is a little bit of overhead associated with keeping track of mail messages. This is primarily the "next" pointers in a linked list of messages. Rather than keep an arbitrary-sized list of mail messages for each mailbox (which is limiting and inefficient), we require the use of "envelopes". Envelopes contain internal accounting data that is needed to keep mail messages in lists, but they also contain some useful information, like the sender of the message.

The user can decide whether to use static envelopes (stored on the stack or globally) or dynamically allocated envelopes (from the heap). It is a good practice to allocate envelopes just before they are sent, and then let the receiver free the envelope memory after he has read the message. Coincidentally, this practice mirrors what we do in real life (sender buys, receiver throws away).

A task may have one mailbox, or it may have many mailboxes, or it may have no mailboxes. There are cases where it is useful to have more than one mailbox. For example, a serial port driver may have one mailbox for traffic to pass along the line and a separate mailbox for flow control messages (you would not want a flow control message to get stuck behind lots of data -- you would want that message to be received immediately). In most cases, however, one mailbox per task is sufficient. To make addressing easier, the first mailbox that a task initializes is it's "primary" mailbox. A task can change which mailbox is considered to be its primary mailbox.

Messages can be sent to a specific mailbox, or they can be sent directly to a task (in which case it is simply delivered to the task's primary mailbox). There are two "send" functions which cover these two options.

So what kind of messages can we send? Most of the time, we are sending simple instructions from one task to another. "I am alive", or "please scan the keyboard". Other times, we need to send a lot of data. Presto sends two things in each mail message: an integer and a pointer. For simple messages, the integer will suffice (you can leave the pointer NULL). For more complex scenarios, the pointer can give the location of a structure that has the needed information. This pointer can point to static (global) memory or to dynamic memory -- it is up to the user to define a protocol for ownership of memory, and who frees what. Again, a good practice is to let the sender allocate and the receiver free.

Mail API

// data types

```
typedef KERNEL_MAILBOX_T      PRESTO_MAILBOX_T;

typedef KERNEL_ENVELOPE_T     PRESTO_ENVELOPE_T;

typedef KERNEL_MAILMSG_T      PRESTO_MAILMSG_T;

typedef KERNEL_MAILPTR_T      PRESTO_MAILPTR_T;
```

// mailboxes, etc

```
extern void presto_mailbox_create(KERNEL_TASKID_T tid, KERNEL_MAILBOX_T * box_p,

extern void presto_mailbox_init(PRESTO_MAILBOX_T * box_p, PRESTO_TRIGGER_T trigge

extern void presto_mailbox_default(PRESTO_MAILBOX_T * box_p);
```

// sending and receiving

```
extern void presto_mail_send_to_box(PRESTO_MAILBOX_T * box_p, PRESTO_ENVELOPE_T *

extern void presto_mail_send_to_task(PRESTO_TASKID_T tid, PRESTO_ENVELOPE_T * env

extern PRESTO_ENVELOPE_T * presto_mail_get(PRESTO_MAILBOX_T * box_p);

extern PRESTO_ENVELOPE_T * presto_mail_wait(PRESTO_MAILBOX_T * box_p);
```

// envelopes

```
extern PRESTO_MAILMSG_T presto_envelope_message(PRESTO_ENVELOPE_T * env_p);

extern PRESTO_MAILPTR_T presto_envelope_payload(PRESTO_ENVELOPE_T * env_p);

extern PRESTO_TASKID_T presto_envelope_sender(PRESTO_ENVELOPE_T * env_p);
```

Timers

This module implements a simple timer system.

Timers can be either one-shot or repeating. They are specified by an initial delay and then a period. Timers can be started or stopped.

When a user wants to use a timer, he simply calls the start function, which requires a delay, a period and a trigger. When the timer expires, the trigger is asserted on process which started it. The user may then wait for that trigger.

Since it is very common to simply delay for a little while, there is a presto_timer_wait function which combines the timer declaration (on the stack), the start with no repeat, and the wait.

TODO - Be smart about when to issue a timer interrupt. Calculate how long until the next timer expires, and then don't bother interrupting until then. This problem is harder than it seems at first!

Timer API

// data types

```
typedef KERNEL_TIMER_T        PRESTO_TIMER_T;

extern void presto_timer_start(PRESTO_TIMER_T * timer_p, PRESTO_INTERVAL_T delay,

extern void presto_timer_wait(PRESTO_INTERVAL_T delay);

extern void presto_timer_stop(PRESTO_TIMER_T * timer_p);
```

Semaphores

This module implements a simple semaphore mechanism.

These semaphores are used when one resource is shared among many users. The semaphore should be initialized at the start of execution. Then, when a user wants to use the resource, he does a presto_semaphore_wait() on that semaphore. His task will be blocked until the resource becomes available.

If there are more than one resource (example: a pool of three printers), then the semaphore can be initialized as a "counting" semaphore (more than one resource) rather than a "binary" semaphore (one resource).

One common problem with resource locking is "priority inversions." To avoid this problem, this module supports priority inheritance, a practice where a running task is promoted to the priority of the highest waiter.

A second problem with resource locking is "deadlock". The only way to avoid deadlock is to pay special attention to the order in which multiple resources are reserved (do not let one task reserve A and then B, while a second task reserves B and then A).

Currently, there is an arbitrary maximum to the number of tasks that can request a semaphore at the same time. This maximum is specified by the constant PRESTO_SEM_WAITLIST in configure.h. This constant tries to find some middle ground between the need to make the semaphore data stucture small in size, and the need to support "popular" resources. Ideally, we would use a more dynamic scheme. In practice, this constant works just fine. If we want to be safe at the expense of memory, we can set

PRESTO_SEM_WAITLIST to PRESTO_KERNEL_MAXUSERTASKS.

Semaphore API

// data types

```
  typedef KERNEL_SEMAPHORE_T    PRESTO_SEMAPHORE_T;

  extern void presto_semaphore_init(PRESTO_SEMAPHORE_T * sem_p, short resources, BO
extern void presto_semaphore_init(PRESTO_SEMAPHORE_T * sem_p, short resources);

  extern BOOLEAN presto_semaphore_request(PRESTO_SEMAPHORE_T * sem_p, PRESTO_TRIGGE

  extern void presto_semaphore_release(PRESTO_SEMAPHORE_T * sem_p);

  extern void presto_semaphore_wait(PRESTO_SEMAPHORE_T * sem_p);
```

Dynamic Memory

This module implements a simple memory pool for fast dynamic memory allocation. It is flexible -- it can be configured to use several pools of fixed sized "items" by changing a handful of constants in configure.h.

There are two main data structures involved with keeping track of memory allocation. There are "pool" structures, which keep high level information about a set of similar memory items (how big are the items, how many are currently used, etc). And then there are the memory items themselves, which each keep track of one block of allocated memory.

The memory item structres are stored interleaved with the actual memory that is allocated. This makes it possible to find a specific memory item, given only a pointer to the allocated memory.

Here is a simple example of the data strctures. mempools: { 5 items, 3 bytes each, A-E }, { 4 items, 7 bytes each, F-I } membytes: A111B222C333D444E555F6666666G7777777H8888888I9999999

In this example, there is a memory item A which precedes three bytes of allocatable memory. The structure in A contains information about those three bytes: is it being used, which pool does it belong to, how many bytes were actually requested (less than three?). When the user asks for up to three bytes, he is given a pointer to the "111" area. When he returns the memory to the pool, he gives us back the same pointer. Using this pointer, we can go backwards a few bytes and see the structure A, and we can return the memory to the pool where it belongs.

Dynamic Memory API

```
  extern BYTE * presto_memory_allocate(unsigned short requested_bytes);

  extern void presto_memory_free(BYTE * free_me);
```

Design considerations

A typical system

Startup sequence and initialization