Using the RISC-V PMP (Physical Memory Protection) with an embedded RTOS to achieve process separation and isolation

JEAN J. LABROSSE, SOFTWARE ARCHITECT
Introduction

Author
μC/OS series of software and books
Numerous articles and blogs

Lecturer
Conferences
Training

Entrepreneur
Micrium founder (acquired by Silicon Labs in 2016)

Embedded Systems Innovator
Embedded Computer Design Innovator of the Year award (2015)

Jean.Labrosse@SiLabs.com
Distinguished Engineer and Software Architect
What Is an RTOS? - Multitasking

- Software that **manages the time** of a CPU
  - Application is split into multiple tasks
  - The RTOS’s job is to run the **most** important task that is ready-to-run

- An RTOS provides **Services** to your application
  - Task management
  - Time management
  - Resource management
  - Message passing
  - Soft Timer Management
  - Etc.
What Is an RTOS? – Preemptive Scheduling

```c
void Low_Prio_Task (void)
{
    Task initialization;
    while (1) {
        Setup to wait for event;
        Wait for event to occur;
        Perform task operation;
    }
}

void ISR (void)
{
    Entering ISR;
    Perform Work;
    Signal or Send Message to Task;
    Perform Work;  // Optional
    Leaving ISR;
}

void High_Prio_Task (void)
{
    Task initialization;
    while (1) {
        Setup to wait for event;
        Wait for event to occur;
        Perform task operation;
    }
}
```
What Is an RTOS? – Tasks

- Each task:
  - Is assigned a **priority** based on its importance
  - Has its own set of **CPU registers** (**Thinks** it has the CPU all to itself)
  - Requires a **stack**
  - Manages its own variables, arrays and structures
  - Possibly manages I/O devices
  - Is typically an infinite loop **waiting** for an event:
    ```c
    void Task (void)
    {
        Task initialization;
        while (1) {
            Wait for event to occur;
            Perform task operation;
        }
    }
    ```
  - Contains mostly the application code
Without a PMP, RTOS tasks run in MACHINE-mode

- Access to all resources
- Done for performance reasons

**Drawbacks:**

- Reliability of the system is in the hands of the application code
  - ISRs and tasks have *full* access to the memory address space
  - Tasks *can* disable interrupts
  - Task stacks can overflow *without* detection
  - Code *can* execute out of RAM
    - Susceptible to code injection attacks
    - A misbehaved task can take the whole system down
  - Expensive to get safety certification for the whole product
What is an RTOS? – Context Switch (without a PMP)
Tasks are grouped by processes
- Can have multiple tasks per process
- ISRs have full access to memory
- Would be very complex otherwise

Benefits:
- Memory of one process is not accessible to other processes
  - Unless they share a common memory space
  - Some processes might not need to be safety certified
    - Less expensive and faster time-to-market
- User tasks can’t disable interrupts
- Task stack overflows can be detected by the PMP
PMP – Each Task requires a Process table

| CSR #0x3A0 | pmpcfg0 | 3 | 2 | 1 | 0 |
| CSR #0x3A1 | pmpcfg1 | 7 | 6 | 5 | 4 |
| CSR #0x3A2 | pmpcfg2 | 11 | 10 | 9 | 8 |
| CSR #0x3A3 | pmpcfg3 | 15 | 14 | 13 | 12 |
| CSR #0x3B0 | pmpaddr0 |
| CSR #0x3B1 | pmpaddr1 |
| CSR #0x3B2 | pmpaddr2 |
| CSR #0x3B3 | pmpaddr3 |
| CSR #0x3B4 | pmpaddr4 |
| CSR #0x3B5 | pmpaddr5 |
| CSR #0x3B6 | pmpaddr6 |
| CSR #0x3B7 | pmpaddr7 |
| CSR #0x3B8 | pmpaddr8 |
| CSR #0x3B9 | pmpaddr9 |
| CSR #0x3BA | pmpaddr10 |
| CSR #0x3BB | pmpaddr11 |
| CSR #0x3BC | pmpaddr12 |
| CSR #0x3BD | pmpaddr13 |
| CSR #0x3BE | pmpaddr14 |
| CSR #0x3BF | pmpaddr15 |

**Application Shutdown Callback**
RTOS with PMP – Context Switch – Only the OS can update the PMP
RTOS with PMP – User tasks run in USER-mode

**RTOS Jump Table**

(Allowed RTOS Services)

<table>
<thead>
<tr>
<th>N</th>
<th>RTOS Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OSSemPost()</td>
</tr>
<tr>
<td>1</td>
<td>OSSemPend()</td>
</tr>
<tr>
<td>2</td>
<td>OSQPost()</td>
</tr>
<tr>
<td>3</td>
<td>OSQPend()</td>
</tr>
<tr>
<td>4</td>
<td>OSMutexPost()</td>
</tr>
<tr>
<td>5</td>
<td>OSMutexPend()</td>
</tr>
<tr>
<td>6</td>
<td>OSTimeDly()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>OSVersion()</td>
</tr>
</tbody>
</table>

**.USER-mode (non-privileged) - MACHINE-mode Handler**

- **Cannot disable interrupts**
- **Cannot change the PMP settings**

**Privileged Code**

- **Can disable interrupts**
- **Can change the PMP settings (but should not)**

**RTOS**

(MACHINE-mode)

**CPU + FPU + PMP**
RTOS with PMP – Setting regions
RedZone:
- Small area at the bottom of each task stack
- Typically 32 bytes
  - Removes from available task stack
  - Larger zone is better but, more wasteful
- CPU exception if data is pushed into that area
- Not guaranteed to catch every overflows!
What happens when a task accesses data outside a valid region?

Based on the region’s attributes, the PMP issues one of three types of an exception:

- *Instruction Access* Fault (i.e. *execute*)
- *Load Access* Fault (i.e. *read*)
- *Store Access* Fault (i.e. *write*)

What can we do when one of these faults is detected?

- Depends greatly on the application
- The RTOS should **save** information about the offending task
  - To help developers correct the problem
- The RTOS should provide a **callback** function for each task
  - To allow the application to perform a *Controlled Shutdown* sequence
    - e.g. Actuators to be placed in a safe state
What can we do when a fault is detected?

- Report the fault:
  - To a display?
  - To a storage device (i.e. file system)
  - Through a communications port?
  - Sound an alarm?
  - Etc.

- Terminate the offending task:
  - Do we also need to terminate other tasks associated with the process?
  - What happens to the resources owned by the task(s)?

- Restart the application
RTOS with PMP - Recommendations

- As a MINIMUM: clear the X (eXecute) bit
  - Except possibly for code that runs software updates
- As a MINIMUM: use a region for stack overflow detection
  - i.e. RedZone
- Run the application in USER-mode
- ISRs should have full access to the memory
  - Greatly simplifies the ISR design
  - But ISRs should be kept short
- Limit peripheral access to its process
- Reduce inter-process communications
  - Processes should be isolated from one another
- Avoid global heap
  - Virtually impossible to setup the PMP for a global heap
- Limit the RTOS APIs available to the user
  - Prevent creating and deleting tasks in USER-mode
- Allocate RTOS objects in RTOS space
  - Processes can provide references to these objects
- Determine what to do when you get a PMP fault
  - Callback to execute an optional shutdown sequence
- Have a way to log and report faults
  - Helps developers correct issues

As a MINIMUM: use a region for stack overflow detection

- i.e. RedZone
Tasks are grouped by process
- A process can consist of one or more tasks

Each task must define a process table
- The process table defines regions
  - Each region gives permission to access a range of memory or I/O space
  - The RTOS loads the process table into the PMP during a context switch
    - Adds overhead

User code gets RTOS services through an MACHINE-mode handler
- Adds overhead

ISRs have full access to memory

You need to determine what happens when you get a PMP fault

The developer is responsible for creating the process tables and splitting the memory in regions
- A tedious job!
References – Websites, Videos & Articles

- **Silicon Labs**:  
  - Micrium OS Kernel (i.e. RTOS) **free** with Silicon Labs MCUs  
  - [www.Silabs.com](http://www.Silabs.com)

- **Micrium (a Silicon Labs Business Unit)**:  
  - μC/OS-II and μC/OS-III RTOS and middleware  
  - Home of μC/Probe  
  - [www.Micrium.com](http://www.Micrium.com)

- **Getting Started with Micrium OS, 10 Episode Series**  
  - [https://www.youtube.com/playlist?list=PL-awFRrdECXu9I7ybAI5tEgwn7BQF6N56](https://www.youtube.com/playlist?list=PL-awFRrdECXu9I7ybAI5tEgwn7BQF6N56)

- **Micrium, Internet of Things**  
  - [https://www.micrium.com/training/videos/#foobox-3/0/SDJVFr4VUHA](https://www.micrium.com/training/videos/#foobox-3/0/SDJVFr4VUHA)

- **Using an MPU with an RTOS, 4 Parts Series**  
Thank you!
RTOS with PMP – Creating Process Tables

- Creating process tables is the most difficult part of using an PMP
  - Requires using `#pragma`s in C code to define *Regions*
    - Toolchain specific
    - Might mean editing existing code files to add `#pragma`
  - Tedious manual effort because of module dependencies
THANK YOU

https://tmt.knect365.com/risc-v-summit