Previous

- RTX Task and Time Management

Inter-Task Communication

- Inter-task communication primitives are of crucial importance to OS design
- Several mechanisms or primitives are provided by RTX
  - Events
  - Mutex
  - Semaphore
  - Mailbox
- The first three provide **synchronization**
- The last one provides **data communication**
- We will have a close look at each one of them one by one.

Events

- An effective alternative to busy-waiting
- Trigger task A when event X happens (event flag set by other tasks)
- Each task has 16 private event flags, corresponding information is stored in the task's TCB
- Ask the task to wait for events and its state will turn to WAIT_AND or WAIT_OR, depending on the version of the event wait call.
- Once the required event flags are set, the task will be put back to the READY state and wait for the scheduler to dispatch it
- Flags can also be cleared by other tasks

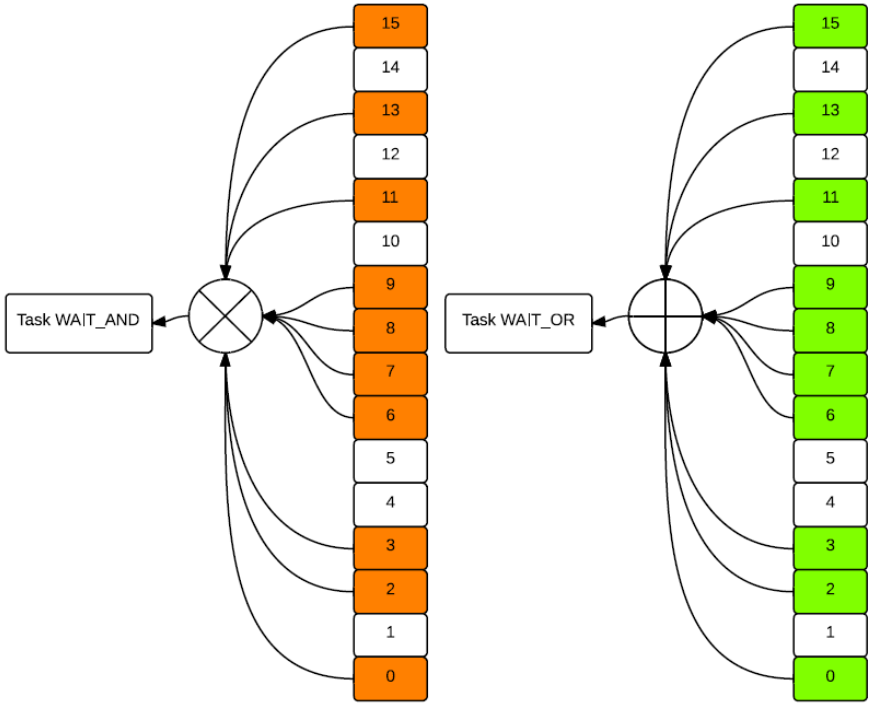
And & Or Events

- Self-explanatory function definitions

  OS_RESULT os_evt_wait_and (U16 wait_flags, U16 timeout);

  OS_RESULT os_evt_wait_or (U16 wait_flags, U16 timeout);

- Task will only monitor the flags specified by *wait_flags*
  - Using the *os_evt_wait_and* function, all monitored flags must be set to trigger the task
  - Using the *os_evt_wait_or function*, setting any monitored flag wil trigger the task
- You can set *timeout* as well, task will be put back to READY after timeout period even if flags are not set
  - As usual, based on timer tick value
  - 0xFFFF means infinite timeout



arm

After Event

- Set the event flags by calling *os_evt_set(0xABCD, taskID);* in another task.
- If event is triggered,
  - Specified flags will be cleared if set
  - *evt_wait* returns the *OS_RESULT*
    - *OS_R_EVT*, if the specified flags have been set
    - *OS_R_TMO*, timeout
    - For or event, you may need to know which exact flags have been set, do the following right after the *evt_wait_or*:
      - *which_flag = os_evt_get();*
      - Then take according steps based on the bit pattern of *which_flag*
- Clear flags by calling *os_evt_clear(0xABCD, taskID);* in another task before it triggers the blocked task.

Interrupt and Event

- As mentioned before, interrupts will “interrupt” the OS if they are not handled properly.
- *systick* is also a form of interrupt, older architectures will disable interrupts while interrupted - less of an issue for Cortex-M.
- Rule of thumb:
  - Use interrupt to set event flags
  - Then do the lengthy job in a task!
  - *void isr_evt_set (U16 event_flags, OS_TID task_id);*
  - Of course, do not forget the IRQ routine stuff (clear interrupt bits for example)

Mutex

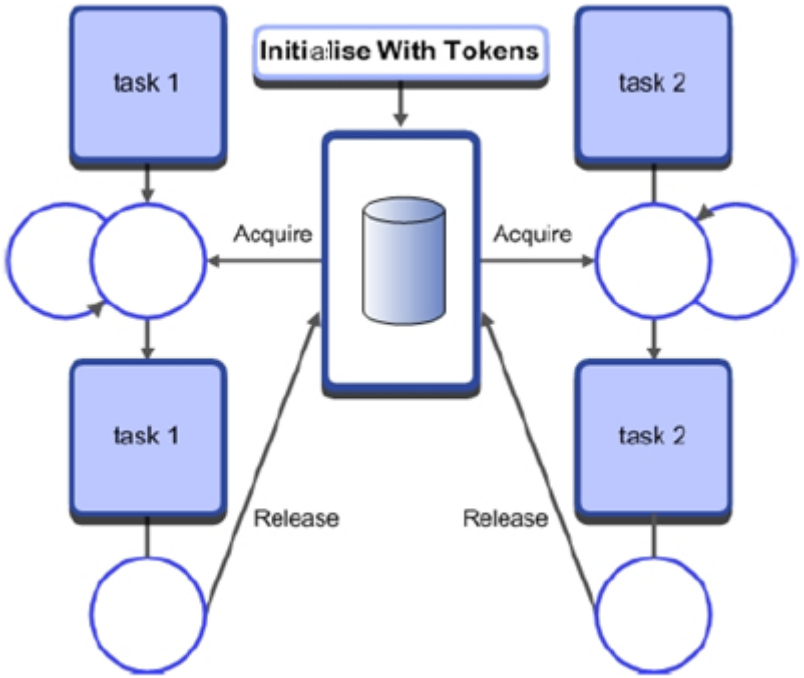
- A mutex, in many ways, is a binary semaphore.
- Two major purposes in RTX:
  - Synchronize tasks
  - Protect shared variables or resources
- RTX mutex has the priority inheritance scheme built in but has no priority ceiling protocol, so be really careful if your task involves more than one mutex.
- Remember to release the mutex after use. RTX will not release the mutex on task deletion – other tasks will never be able to acquire it again!
- An alternative way: use the lock and unlock functions.
  - Task acquired a mutex can still be interrupted and blocked
  - Whereas a locked task will not be preempted by the OS (more control over everything)
  - But in most cases, mutex is what you want.

Mutex Routines

- Three related functions and one type only
- Declare a mutex object first: *OS_MUT mutexID;*
- Then initialize the mutex through *os_mut_init (mutexID);*
- Grab the mutex through *os_mut_wait (mutexID, timeout);*
  - *timeout*, yes, its usage is identical to that in *evt_wait*
  - Return values:
    - OS_R_MUT: has to wait and task is put to WAIT_MUT state and will be put back to READY if the mutex is later on released
    - OS_R_TMO: timeout
    - OS_R_OK: has the mutex and task can continue
- After the critical section, release the mutex through *os_mut_release (mutexID);*
  - What if the task does not have the mutex?
  - Mutex is not semaphore!

Semaphore

- Recall that a semaphore is a container of a number of tokens.
- Acquire a token first, then access a resource.
- To finish with the resource, return the token.
- Also used to synchronize tasks or protect variables and resources.
- Very sophisticated and comprehensive ways of using semaphores.
- Unlike mutex, you may destroy a task with unreleased tokens (depends on how you use your semaphore).
- Simple routine APIs, but frequently misused!



arm

Semaphore Routines

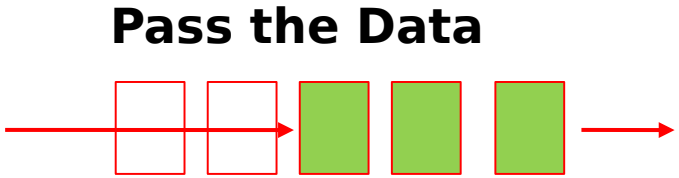
- Very similar to the mutex. Again, three related functions and one type only.
- Declare a semaphore container first: *OS_SEM semID;*
- Then initialize the semaphore through *os_sem_init (semID, unsigned token_count);*
  - *token_count*, the initial number of tokens
- Grab a token through *os_sem_wait (semID, timeout);*
  - *timeout*, as expected, just like *mut_wait* and *evt_wait*.
  - Return values:
    - OS_R_SEM: has to wait until a semaphore is available, task is put to WAIT_SEM state and will be put back to READY if some task sends a semaphore.
    - OS_R_TMO: timeout
    - OS_R_OK: has the token and task can continue
- Return/create a token in the semaphore container through *os_sem_send (semID);*
  - Can a task return(create) a token even if it has no token?
  - Semaphore is not a mutex!
  - Its counterpart for ISR : *isr_sem_send(semID);*

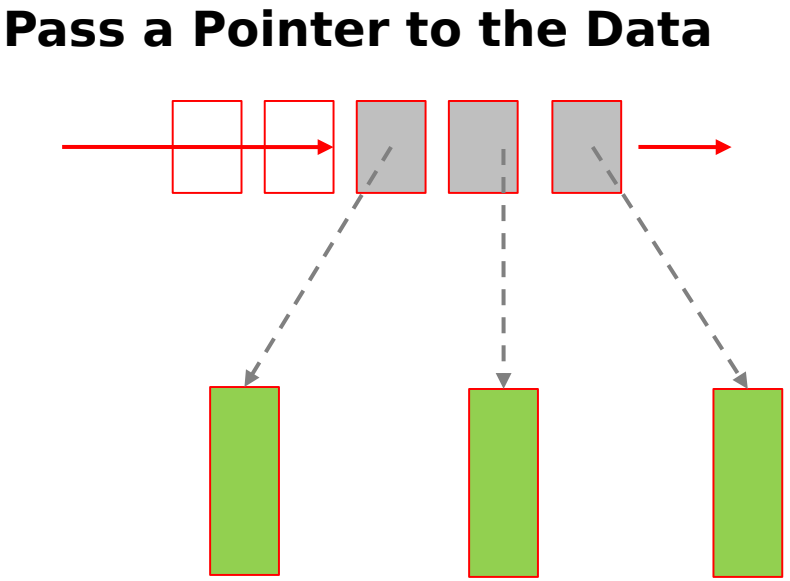
Mutexes vs. Semaphore

- Semaphore
  - Used for signaling from tasks or ISRs to waiting tasks
  - Can be initialized to 0, meaning “The event hasn’t happened yet”
- Mutex
  - Used to ensure mutually exclusive access to a shared object
  - Is a binary semaphore with priority inheritance and some other changes
  - Is initialized to 1, meaning “The object isn’t being used now”
- Common pitfall
  - Semaphore handles several copies of equivalent resources whereas mutex handles one?
  - Semaphore does not keep track of the order of access – effectively treating all resources identically! If your multiple copies of equivalent resources are not identical,
    - Access and release order matters
    - Use multiple mutexes instead of semaphores!

Mailboxes and Messages in RTX

- Sharing bigger chunks of information between tasks
- A mailbox is a queue of user-defined length
- Each queue element (message) is four bytes long, can be used in two ways
  - Can hold the data itself (up to 32 bits)
  - Can hold a pointer to larger objects
- When passing larger objects, need to dynamically allocate memory per object, free when done processing





Creating Mailboxes

- Create mailbox
  - *os_mbx_declare(mailbox_name, mail_slots)*
  - Declares an array (*mailbox_name*) of 32-bit unsigned ints
  - One element per message (*mail_slots*)
  - Four extra elements for mailbox management
- Initialize the mailbox
  - *os_mbx_init(&mailbox_name, sizeof(mailbox_name))*
  - Takes pointer to mailbox and size in *bytes*

In RTL.h:
#define os_mbx_declare(name,cnt) U32 name [4 + cnt]

Sending Messages

- Prepare message if passing a pointer to the data
  - May need to allocate memory (if object doesn't exist)
- Send the message
- *os_mbx_send(&mailbox, msg_ptr, timeout)*
  - Send message pointed to by *msg_ptr*
  - What if mailbox is full?
  - *os_mbx_send* will block (task will sleep) until space becomes available
  - *timeout* again
  - Return code: *OS_R_OK* and *OS_R_TMO*
- *isr_mbx_send(&mailbox, msg_ptr)*
  - ISRs use this version
  - Use *isr_mbx_check()* to find number of free message entries in mailbox
  - If mailbox is full, the message is discarded and *os_error()* function is called, placing system in infinite loop by default

Receiving Messages

- *os_mbx_wait(&mailbox, &msg_ptr, timeout)*
  - Provides a pointer to the next message in the specified mailbox
  - Must free the message's memory after using it
  - Will block if the mailbox has no message
  - *timeout* again
  - Return code:
    - *OS_R_MBX*: has to wait to get message
    - *OS_R_OK*
    - *OS_R_TMO*
- *isr_mbx_receive(&mailbox, &msg_ptr)*
  - Provides a pointer to next message
  - Return value indicates result
    - *OS_R_MBX*: message was in mailbox and was read
    - *OS_R_OK*

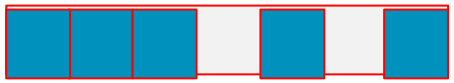
Dynamic Memory Allocation

It doesn't fit anywhere!

Single pool handling all size requests

Here!

Pool for 32-byte requests



Pool for 16-byte requests

- Allocating and freeing different sizes of memory can cause fragmentation, so free space is distributed in fragments which are too small to use
- Solution is to create a separate memory pool for each size of data to be allocated

RTX Dynamic Memory Allocation

- RTX has powerful fixed memory block memory allocation routines - box
- “box” = memory pool of fixed-size blocks
- Thread safe and re-entrant
- Functions
  - *_declare_box(box_name, block_size, block_count)* allocates memory statically (declares an array called *box_name*)
  - *_init_box(box_name, box_size, block_size)* initializes the box
  - *_alloc_box(box_name)* returns a pointer to a memory block, or NULL on failure
  - *_calloc_box(box_name)* returns a pointer to a memory block (initialized to 0), or NULL on failure
  - *_free_box(box_name, block)* frees up the block in that pool
- Better to use these for fixed size messages
- Variable length memory allocation (malloc() and free() from stdlib.h) functions are not re-entrant! Use lock and unlock to disable context switch!

A Working Example

```
os_mbx_declare (MsgBox, 16); /* Declare an RTX
mailbox */
U32 mpool[16*(2*sizeof(U32))/4 + 3]; /* Reserve a
memory for 16 messages */


__task void rec_task (void);


__task void send_task (void) {
    /* This task will send a message. */
    U32 *mptr;
    os_tsk_create (rec_task, 0);
    os_mbx_init (MsgBox, sizeof(MsgBox));
    mptr = _alloc_box (mpool); /* Allocate a memory
for the message */
    mptr[0] = 0x3215fedc; /* Set the message content.
*/
    mptr[1] = 0x00000015;
    os_mbx_send (MsgBox, mptr, 0xffff); /* Send a
message to a 'MsgBox' */
    os_tsk_delete_self ();
}
__task void rec_task (void) {
    /* This task will receive a message. */
    U32 *rptr, rec_val[2];
    os_mbx_wait (MsgBox, (void**)&rptr, 0xffff); /*
Wait for the message to arrive. */
    rec_val[0] = rptr[0]; /* Store the content to
'rec_val' */
    rec_val[1] = rptr[1];
    _free_box (mpool, rptr); /* Release the memory
block */
    os_tsk_delete_self ();
}


int main (void) {
    _init_box (mpool, sizeof(mpool), sizeof(U32));
    os_sys_init(send_task);
}
```


arm

Sharing Data Safely

- Preemption and interrupt could contaminate data:
  - One writer and at least one reader scenario
    - Data overwritten partway through being read - Writer and reader might interrupt each other
  - More than one writer, at least one reader scenario
    - Data overwritten partway through being read - Writer and reader might interrupt each other
    - Data overwritten partway through being written - Writers interrupt each other
- Is the read/write operation indivisible/atomic?
  - Yes, then problem solved – but what if not?
- Race condition
  - Anomalous behaviour due to unexpected critical dependence on the relative timing of events
  - Depends on the *relative timing* of the read and write operations
- Critical section
  - A section of code which creates a possible race condition
    - Any access to a shared data structure is a critical section of code
  - Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use

Data Race Bug: Timestamp Data Structure

- System
  - TimerVal structure tracks time and days since some reference event
  - TimerVal's fields are updated by periodic timer ISR
- Problem
  - An interrupt at the wrong time will lead to wrong data in DT

```
void GetDateTime(DateTimeType * DT){
 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
}
```

- Failure Example
  - TimerVal is {10, 23, 59, 59}
  - Task code calls GetDateTime(), which starts copying the TimerVal fields to DT: day = 00, hour = 23
  - A timer interrupt occurs, which updates TimerVal to {11, 0, 0, 0}
  - GetDateTime() resumes executing, copying the remaining TimerVal fields to DT: minute = 0, second = 0
  - DT now has a time stamp of {10, 23, 0, 0}.
  - ***The system thinks time just jumped backwards one hour!***

Atomicity, ARM ISA and Shared Memory

- ARM is a Load/Store architecture – variables can only be modified in registers, not in memory
- Any memory-resident variable must be accessed with at least three instructions: load, modify, store.
  - This creates a **critical section** from the load instruction to the store instruction (inclusive)
  - It's not just multi-element data structures – **ANYTHING** in memory (even bytes or words) is vulnerable
- Tasks communicating with shared memory are vulnerable to **race conditions**
- Any variables used in shared memory communication must be protected

Reentrancy and Data Sharing

- A naive implementation itoa() function
  - Converts an integer to a text string (char array)
  - Space for only one string is allocated, so itoa() is non-reentrant.
  - **Not ensuring mutually exclusive global resource use makes the function non-reentrant**
- Various fixes possible
  - Could change itoa() so that the calling function must allocate space for string
  - Could change itoa() to store the buffer locally
  - Could protect the array with a semaphore to ensure mutual exclusion
  - Could prevent task switching (and maybe even interrupts) in critical sections for global data
- itoa() is called by multiple tasks and is representative of application library code (math libraries, error handlers, I/O drivers)
- **Observation: sometimes data sharing problems are hidden in non-reentrant shared functions**
- Write re-entrant code
  - Code which can have multiple simultaneous, interleaved, or nested execution instances which will not interfere with each other
  - This is important for multi-threaded code, recursive functions, and interrupt handling
- If each instance has its own data, the code is reentrant (e.g. using own stack frame and restoring all relevant processor state)

General Solutions Based on RTX

- Use the primitives introduced to avoid data race
- Single Writer
- Buffer data so reader and writer don’t access the same copy
  - Do-it-yourself buffering
  - Mailbox and message!
- Multiple writers
- Make sure only one writer accesses the shared variable
  - Use Mutex
  - Or disable preemption/interrupt (Disable Cortex-M interrupt or lock/unlock RTX)
- Also double check your tasks, better to make them re-entrant unless you have good reasons not to

Next

- Lab
- Performance Evaluation and OS-Aware Debugging