How to choose an RTOS

IAR SYSTEMS

Agenda

- What is an RTOS?
  - Definition
  - Free vs. Commercial RTOSs
- Benefits of using an RTOS
  - Safety
  - Reliability
  - Rapid development
- What should I look for in an RTOS?
- Common elements of RTOSs
- Example of an RTOS in action
  - Webserver
  - USB Device

What is an RTOS?

- An RTOS is an operating system designed to service the needs of an embedded system in real-time.
- It gives you the ability to access peripherals on the device in a manner similar to the way that you access files in desktop C
- It helps you keep processes in your application from interfering with one another by placing them in tasks
- It gives you modularity to add plug-ins such as a USB Host Stack or a TCP/IP Stack
- It controls inter-task communication through the use of messages and queues
- It can use semaphores to ensure that only one task at a time uses a certain peripheral
- It assigns priorities to tasks according to their importance in the application and switches between them using a scheduling algorithm

Free vs. Commercial RTOSs

- A free RTOS offers a cost-effective solution to manage your code with the benefits previously mentioned:
  - No cost for the RTOS
  - No royalties for products made using the RTOS
  - Easy to try out in your design
- However, free RTOSs have some drawbacks:
  - Not as well-supported as commercial RTOSs
  - Generally not as well-optimized as commercial RTOSs
  - Generally do not have as many plug-ins or bolt-ons as commercial RTOSs

Benefits of using an RTOS

- Scheduling algorithms – although easy to comprehend – are amazingly difficult to write:
  - Latent defects in context switching from one task to another
  - Minimizing context switch latency
  - Priority inversion issues
- Using an RTOS allows you to increase the safety of your code by ensuring that code can execute in deterministic time or meet a certain deadline
- Using an RTOS increases the reliability of your code by encapsulating the individual tasks in such a way that they do not interfere with one another
- By allowing the RTOS to do your scheduling and priority, you simplify the application code that you have to write and speed your time-to-market

What to look for in an RTOS

- Determinism – does the RTOS have published numbers on the minimum, average and maximum cycles its functions require?
- Interrupt latency – how long does it take between the time the interrupt is received by the MCU and the time the Interrupt Service Routine (ISR) begins executing?
- Context switch time – how long does it take the RTOS to switch between the different tasks in the task pool?
- Available plug-ins – USB Host, USB Device, TCP/IP, Bluetooth, SSH, SSL, etc.
- Compatibility of RTOS with your chosen toolchain
- Overall cost of RTOS – initial cost, procuring source code of RTOS, support costs, royalties, maintenance costs

It should be relatively easy to compare these features from one RTOS to another once you have a target and feature set decided.

Common features of RTOSs

Memory allocation

- Usually, most objects are instantiated at run-time, so an RTOS needs the ability to dynamically allocate memory.
- The standard C function `malloc` is not thread-safe, so the RTOS must provide a thread-safe method to replace it.
- Different RTOSs do this in different ways, so if your application does quite a bit of dynamic allocation and recollection of memory, this can have a deleterious impact on your code.
- In general, higher-end commercial RTOSs do the best job of memory allocation
- Be sure to use the thread-safe memory functions provided to you by the RTOS!

Common features of RTOSs

Tasks

- All RTOSs allow you to perform functionality within a task
- It is sometimes difficult to tell if two tasks should be combined into one or vice-versa, similar to deciding whether two functions should be combined into one or vice-versa
- A task has a control block which is a piece of RAM that contains the task's stack and other important information that is written to whenever the task is "switched out" for another task. Similarly, that information is loaded back when the task is "switched in", so the state of each task is saved in the block.
- Different RTOSs have different limitations on the number of tasks that can be created and on the number of different priorities that can be assigned to the tasks. A typical RTOS application uses 10-15 tasks.

Common features of RTOSs

Task schedulers

- Different RTOSs use different scheduling algorithms
  - Co-operative schedulers require the tasks to `yield( )` in order to allow other tasks to execute.
  - Pre-emptive schedulers are more common in RTOSs and may take several forms:
    - Round-robin scheduling, where each task gets a time-slice of the MCU
    - Fixed-priority pre-emptive, which ensures that the next task with the highest priority gets executed
    - Rate-monotonic scheduling, where the task with the shortest duration has the highest priority
  - And others…
- Some schedulers are more susceptible to priority inversion than others, so be aware of deadlocks!

Common features of RTOSs

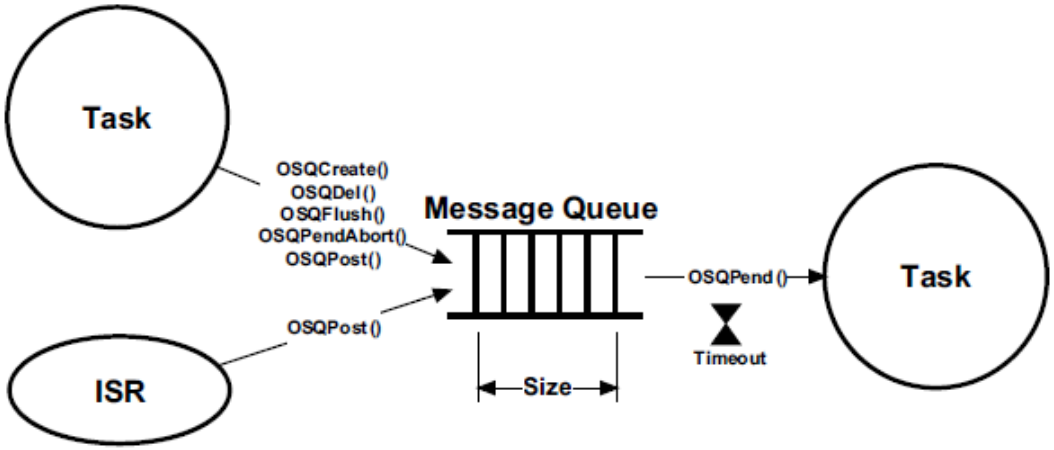
Queues/messages

- These provide a thread-safe manner for inter-task communication so that you don't have one task reading a piece of RAM while another is writing to it
- A message is a fixed-length queue and therefore generally has less overhead associated with processing it
- Different RTOSs place different restrictions on the size of queues as well as how many queues/messages can be created
- These messages are processed in a FIFO manner, so the first queue/message to arrive is the first to be processed
- The RTOS handles the allocation and de-allocation of the queue/message using the same thread-safe memory management previously described, so the efficiency of the manager can impact the performance for copious queues

Common features of RTOSs

Queues/messages

An ISR can only post to the Queue whereas a task allows you to create and modify a queue



Common features of RTOSs

Semaphores

- A semaphore is a flag of sorts that allows things to happen:
  - It may block a task from executing until a condition is satisfied
  - It may block access to a resource like an I2C bus
- A task waiting on a semaphore is said to *pend* until the semaphore allows it to continue.
- Once the semaphore changes, it signals the RTOS to convey the state change to all interested tasks which may cause a pending task to become active
- Semaphores can cause priority inversion where a lower priority task has locked access to a resource that a higher priority task needs – some RTOSs solve this by priority inheritance which temporarily changes the priority of the low priority task to a higher priority so that it can complete and release the resource at which time it is reverted back to a low priority.

Common features of RTOSs

Events/Flags

- A semaphore is a simple flag that allows a piece of code to execute if a resource is available or a condition is true
- In general, events/flags allow the developer to specify a series of conditions that can halt the execution of a piece of code until all the conditions defined by the events/flags are true
- These give a greater degree of flexibility to the developer, but exactly how these are implemented (and how many can be specified) varies from RTOS to RTOS, so it is important to know a priori approximately how many blocks a task may need when you choose your RTOS
- Having a good kernel-aware debugger is key when trying to debug problems related to events/flags!

Common features of RTOSs

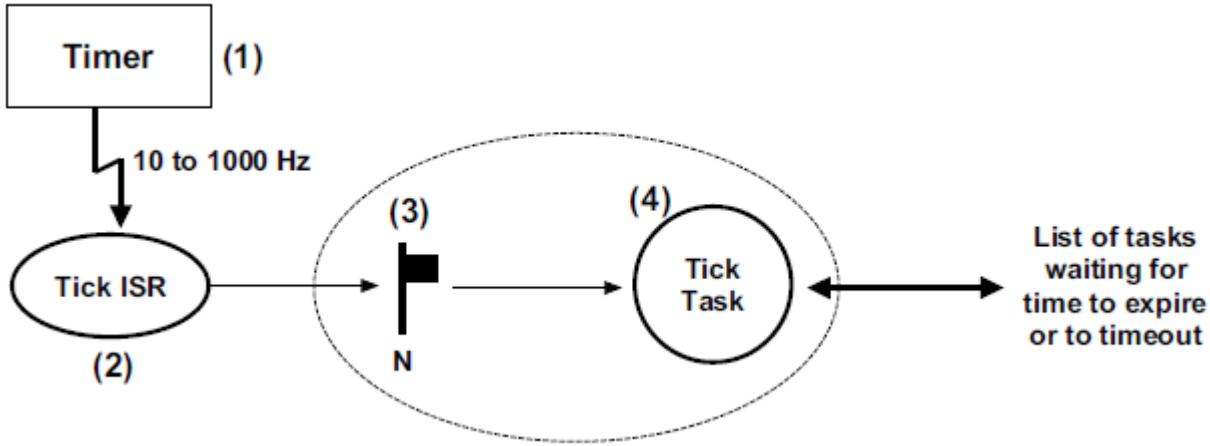
Ticks/Timers

- A tick is often referred to as the “heartbeat” of an RTOS
- It wakes up the kernel and allows it to switch from one task to another
  - A task completion between ticks can also cause a task switch
  - The tick must ***always*** run or the RTOS dies
- The tick is usually set to around 500Hz-1kHz
  - If the tick is set too fast, the RTOS loses performance because it spends all its time switching in the kernel
  - If the tick is set too slow, the RTOS loses performance because it is too slow to respond to stimuli
- Timers allow tasks to delay/wait for a finite period of specified time before the task continues execution
- Timers can also be used to allow tasks to run at set intervals

Common features of RTOSs

Events/Flags

(1) Hardware timer tick
(2) Tick Interrupt Service Routine
(3) OS Service to make tick task “ready”
(4) The tick task



Common features of RTOSs

Hook functions (a.k.a. callback functions)

- A hook function allows a developer to attach a piece of code to a particular function within the RTOS
- These can be handy if the RTOS source code is not available because it is proprietary or because it costs more money
- These can also be handy to ensure that you don't actually touch the RTOS source and inadvertently introduce a defect
- Hook functions can allow you to do pre- or post-processing of information in the RTOS without the overhead of a separate task creation
- Common uses of a hook function include enhancing debugging, key debouncing and enhancing functionality

Common features of RTOSs

Hook functions (a.k.a. callback functions)

For instance, the debouncing of switch inputs may be more simply done by attaching a small piece of code to the periodic heartbeat function rather than creating a separate task to perform this function.

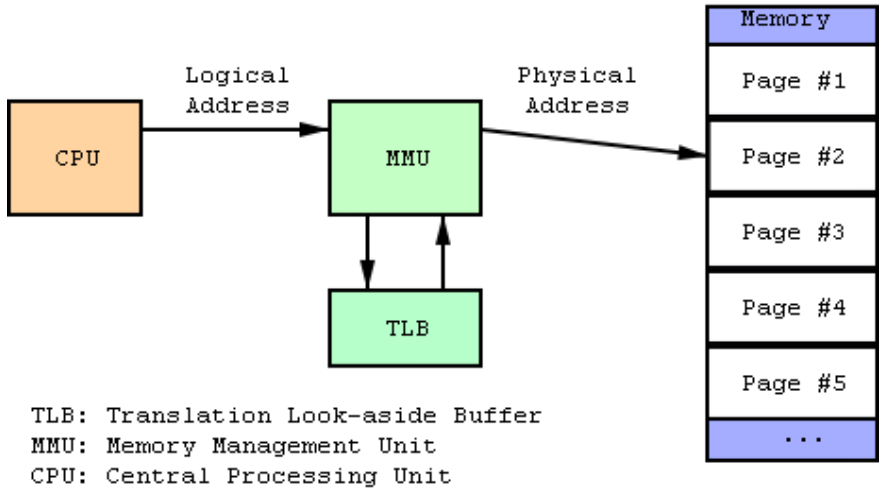
```
int debounce(int SampleA)
{
static int SampleB = 0;
static int SampleC = 0;
static int LastDebounceResult = 0;

LastDebounceResult = LastDebounceResult &
(SampleA | SampleB | SampleC) | (SampleA &
SampleB & SampleC);
SampleC = SampleB;
SampleB = SampleA;
return LastDebounceResult;
}
```

OSTimeTickHook()

Common features of RTOSs

MMU and MPU

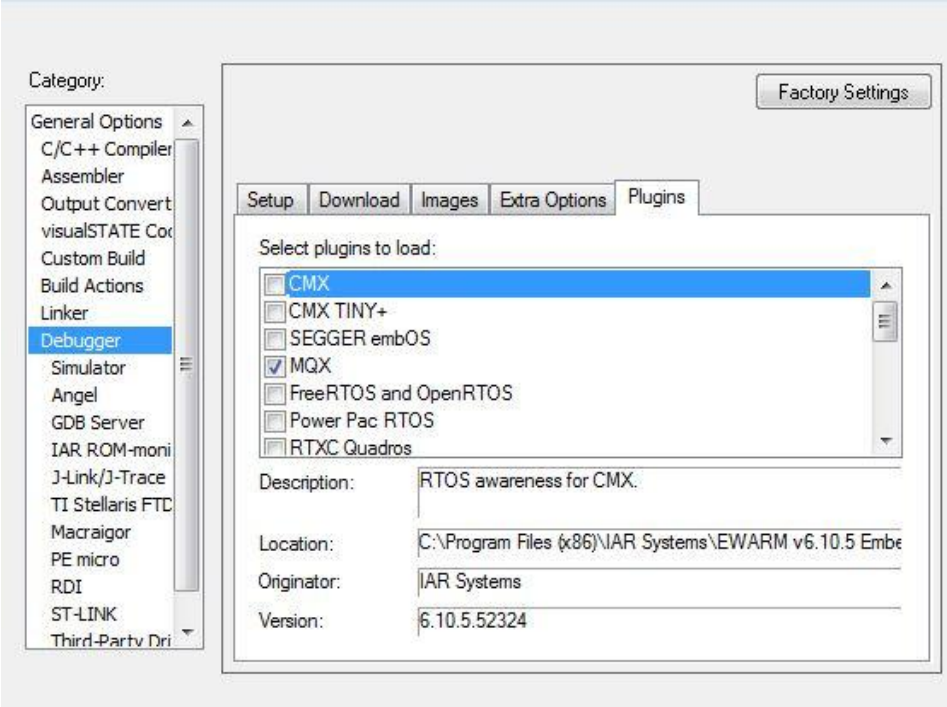
- Stack overflows are easily detected if the processor has a Memory Management Unit (MMU)
- Can be configured to detect when a task attempts to access invalid memory locations
- The MMU normally translates virtual page numbers to physical page numbers via an associative cache called a Translation Lookaside Buffer (TLB)



Using RTOSs in Embedded Workbench

IAR has an RTOS partner program in which RTOS vendors provide plug-ins to make the Embedded Workbench kernel-aware for their RTOS. As such, the plug-ins can look quite different from RTOS to RTOS.

In order to make the C-Spy debugger kernel-aware, select your RTOS in Project-Options-Debugger-Plugins.



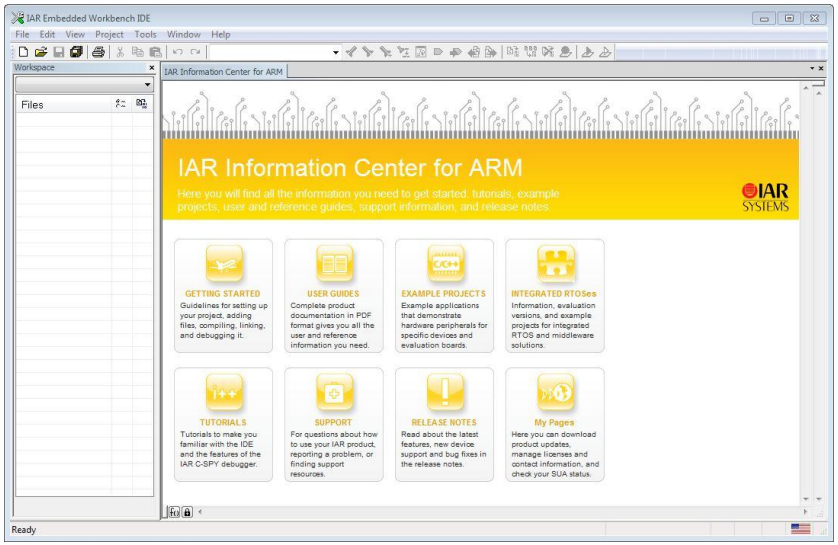
Don’t start from scratch!

RTOSs generally have a rich portfolio of example projects, from the mundane LED blink to a complex webserver that reads HTML files from a USB stick. Be sure to browse the examples to see if one closely matches what your end application is.

Nothing gets you started more quickly than a well-written example!

Demo

- Open up the Embedded Workbench for ARM
- Choose "Integrated RTOSs" from the Information Center
- Check the Project-Options to make sure it is setup for our hardware
- Recompile the code to make sure it compiles with no errors or warnings, then download and debug!



Q&A