NobleProg

Introduction to freeRTOS

Grzegorz Mazur

NobleProg

What is an RTOS?

- Real-Time Operating System – software enabling quasi-parallel execution of multiple programs (tasks/threads) in a microcontroller
- Provides task/thread switching and thread synchronization
- Does not make it impossible to provide deterministic response time to events

µC firmware structures (1)

- Init sequence + event loop
  - Trivial design, only for demo purposes
  - may cause event dropping if >1 event
  - unnecessary energy consumption caused by event polling
- Init + event loop + interrupts
  - May cause difficulties in energy saving with >1 event
  - Better response time for simple events
  - Requires proper decomposition of tasks between ISRs and event loop
  - Care must be taken to ensure that no event is lost during other event processing

µC firmware structures (2)

- Init routine + event handlers (ISRs), without event loop
  - Minimizes energy consumption, fastest event response, requires significant programmer’s experience and mastering the event synchronization via signals/interrupts
  - Not a popular approach (more at https://www.state-machine.com)
- RTOS → multiple event loops
  - Conceptually simple design of a complex firmware with many concurrent event loops, at the cost of RTOS time overhead
  - Does not require paradigm switch from programmers

Program, task, thread (1)

- Program – static notation of an algorithm and data used by it
  - In a single-tasking single-threading system: program == task == thread
- Task/process – an instance of a program in computer’s memory in a multitasking system
  - A task has its own, private memory, not accessible by other tasks
  - Many tasks may be present in a multitasking system; this also includes multiple tasks created from a single program, with separate memory spaces
  - In a single-threading system task == thread

Program, task, thread (2)

- Thread – a running (dynamic) instance of a task/process
  - Threads of a task share the task's memory, thread's memory is not protected against other threads' accesses
  - every thread has its own stack
  - in addition to task's static data used by all threads, a thread may (optionally) have its own static data
  - in simple computers without full memory management (like µCs) there are no isolated tasks (no memory protection between tasks → the whole software is a single task) but there may be multiple threads (like in FreeRTOS)

RTOS for a µC vs. „full” OS

- No task isolation – single big program/task, many threads
  - no memory protection (or basic protection if supported by the µC)
  - static data shared by all threads
  - all threads may use all procedures within a program
- Only thread switching and synchronization
  - Basic RTOS does not contain device drivers, file system nor user console
    - these may be added as optional modules

RTOS and „real time”

- real-time system should guarantee the timely response to events in a specified time, as needed by the controlled object
- it does not mean “immediate response”, it’s merely “guaranteed longest response time”

RTOS and „real time”

- Real-time = fast enough to guarantee timely response, as required by the application
  - Not an immediate response
  - Not always really fast or fastest
- RTOS slows down the event responses
- „If there is an RTOS, there is no real time” ;)
- RTOS slows down the firmware operation but simplifies its design

FreeRTOS

- The most popular small, free of charge RTOS
  - currently owned by Amazon
  - there are tens/hundreds of similar systems available
- Available for >30 architectures
  - Easily portable to new architectures – only single module needs rewriting (ca. 300 lines of code)
- Commercial, paid versions available (support, certification)
  - OpenRTOS – same as FreeRTOS, different license
  - SafeRTOS – modified source, certified for safety-aware applications

Documentation

- Available online – freertos.org
  - Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide
  - Reference Manual

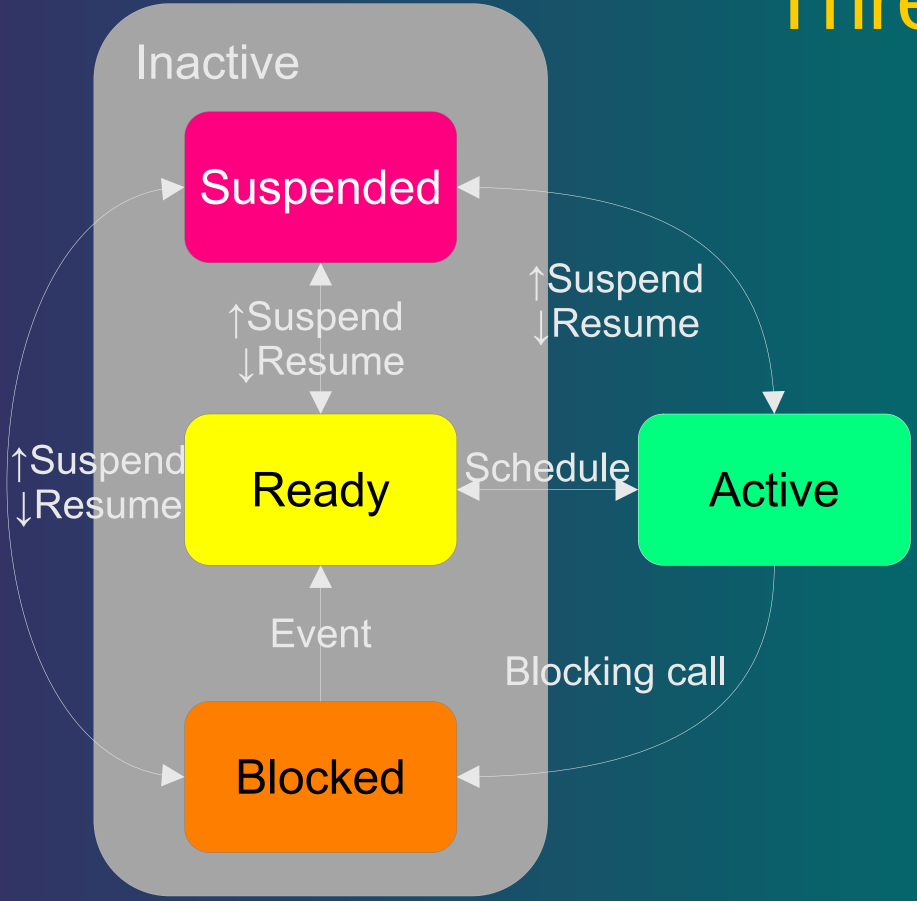
Thread in FreeRTOS

- Fragment of program – procedure with an infinite loop, which may be executed quasi-concurrently with other threads
- Each thread has its private stack
- Other data – static and heap – may be freely shared between threads
- Threads using the same code may optionally have unique static data, kept in a structure
  - thread procedure has an argument, which may be a pointer to this structure
- In native FreeRTOS, thread is called “task”
  - Not quite correct

System timer and time sharing

- Usually FreeRTOS uses preemptive (forced) time sharing
  - configurable option, may be turned off
- Time sharing requires periodic timer interrupt
  - Typ. 1 kHz (may be changed in system configuration)
  - Threads may be switched with this frequency
  - ARM core SysTick Timer is used for this purpose

Thread states



- Active – currently being executed by the processor
- Inactive
  - Ready – may become active (not waiting for an event; the initial state of a thread)
  - Blocked – waiting for an event
  - Suspended – temporarily turned off (not needed in the current system context)

Thread priorities

- FreeRTOS allows for prioritizing the threads
- No. of priority levels available set in system configuration
- No. of priority levels influences the data memory usage
- At least two priority levels must exist
  - there is a system idle task with priority lower than all other tasks
- Avoid differentiating thread priorities unless necessary

Thread scheduling

- Thread may be activated if:
  - It is ready
    AND
  - There is no ready thread with a higher priority
    AND
  - System uses time sharing or the active thread invokes the scheduler (changing its state to Ready) – by calling osThreadYield()

Thread scheduling

- Thread cannot become active if:
  - It is not Ready
    OR
  - There exists an active or ready thread with a higher priority
    OR
  - time sharing is turned off and a thread of the same priority is active

FreeRTOS structure

- FreeRTOS consist of (only) few architecture-independent modules written in C
  - task management - obligatory
  - queue management – almost obligatory
  - optional: software timers, notifications, events, coroutines
- Architecture-specific code is contained in single, compact file – port.c

FreeRTOS configuration

- All the options and parameters may be set via FreeRTOSConfig.h file
- Options enable system functions
- Option activation and parameter values influence the program and data memory usage

FreeRTOS and CMSIS

- CMSIS – standardized, chip vendor-independent API for ARM Cortex microcontrollers, managed by ARM
- There are tens of RTOSes similar to FreeRTOS
  - … and each of them has its own API
- ARM solution – provide uniform API for all RTOSes – CMSIS-RTOS
  - current version: CMSIS-RTOS2 (significantly different from CMSIS-OS/CMSIS-RTOS)
  - Documentation available at: https://arm-software.github.io/CMSIS_5/RTOS2/html/index.html

RTOS in CMSIS

- RTOS functions in CMSIS have names starting with „os”
- function names are standardized – the same application firmware may be compiled and work with different RTOSes
- CMSIS and native RTOS calls may be freely mixed (not recommended for code clarity)
- with CMSIS-RTOS, a single set of functions may be called from threads and ISRs

Thread in FreeRTOS

- A thread is described by:
  - Name (text string – only for debugging purposes)
  - Priority
  - Thread initialization procedure address
  - Thread application stack size (expressed in words, not bytes)
- After creation, the thread is identified by its handle

Thread stack

- Allocated by FreeRTOS during thread creation
- Contains thread local objects related to procedure calls (arguments, return traces, local variables)
- Stores thread context when the thread is not active (*)
  - with ARM, only part of the context is stored on the application stack

Thread stack size

- Declared by programmer (in words, not bytes!)
- Must ensure enough space for local data and register storage with the maximum possible call nesting level and context store for inactive thread (*)
- Insufficient stack space is one of the most common errors in RTOS-based firmware
  - Use the tools available in your IDE, like call stack usage monitor

Thread procedure

- Thread procedure has an optional parameter
  - any type that could be cast to void*
  - it may be a scalar value or a pointer to a structure with multiple parameters
  - single procedure with different parameter values may be used for multiple threads
- return from thread procedure is not allowed
  - The procedure should end with an infinite loop, like for (;;)
    - alternatively it may delete the thread at the end by calling the OS service

Coroutines

- An old, simple approach to running multiple pieces of code using collaborative approach
  - A coroutine explicitly gives up control
- Coroutines share common stack space, so they cannot use local variables
- FreeRTOS provides limited coroutine support
  - Use of coroutines is not recommended
  - Coroutines may be used on small 8-bit microcontrollers with very limited memory

Timers and delays

- Time-dependent operations may be implemented in two ways:
  - using delay functions in a thread code – osDelay(), osDelayUntil()
    - with osDelay(), delay may be bigger than expected (accumulating with each call)
    - osDelayUntil() enables the fixed period operation with jitter but without delay accumulation
  - system software timers – osTimerNew(), osTimerStart()
    - single – calls a routine after specified time
    - Periodic – calls a routine in fixed intervals (period jitter, no delay accumulation)
- A thread or timer routine may be activated later than planned!

Thread communication and synchronization

- Managing thread activity is the basic RTOS functionality
  - activating when ready
  - deactivating when not ready/nothing to do
- sync mechanisms are used to inform the system that a thread may be activated
- Queue is the root, generic sync mechanism in FreeRTOS

Queue in RTOS

- Queue is a buffer which may hold a specified number of elements of a specified type/size
- It is used to pass the data (“messages”) between threads – producers and consumers
- Queues are global – every thread may use a queue
- basic operations: osMessageQueueGet(), osMessageQueuePut()
- with osMessageQueuePut(), producer must wait if a queue is full
- with osMessageQueueGet(), consumer must wait if a queue is empty

Other synchronization mechanisms in FreeRTOS

- Counting semaphore – used for allocation of multiple resources
- Binary semaphore – used to allow for/enable some operation
- Mutex – binary semaphore for allocating a single resource
- flags:
  - thread flags
  - event flags
- (not yet supported in CMSIS-OS) xxx

Counting (multivalue) semaphore

- Used for allocating the resource which may be used by at most n users
  - implemented as a queue of empty elements
    - allocation – get the element from a queue – blocks if value = 0
    - deallocation – put the element – always possible
      - Unlike the standard queue, the semaphore is never “full”
- Allocation: osSemaphoreAcquire()
- Deallocation: osSemaphoreRelease()
  - Unlike osMessageQueuePut(), osSemaphoreRelease() is a non-blocking call (the released semaphore may always be released)

Binary semaphore

- Used to allow the thread to preform some action
- Implemented as single-element queue with an empty element
- Allow/”semaphore up” – osSemaphoreRelease() - called from some other thread or ISR
- start the action (waiting until allowed)/”semaphore down” – osSemaphoreAcquire()

Mutex

- A restricted binary semaphore for allocating a single resource
  - only the thread which claimed the resource may release it
- Must be released by the thread which acquired it
- If the priority of a thread waiting for the mutex is higher than the priority of a thread owning it, the priority of the owner is temporarily raised, so that it may release the mutex
- Allocation: osMutexAcquire()
- Deallocation: osMutexRelease()

Thread flags

- Set of binary semaphores specific to a thread
  - not global, thus faster than semaphores
- A thread may wait for an arbitrary subset of available flags:
  - A single flag
  - One of multiple selected flags
  - All of the specified/selected flags
- When a specified condition occurs, the flags set may be cleared
  - Option, the thread may also selectively clear the flags in a separate operation

Event flags

- Global collections of flags – a crossover of thread flags and binary semaphores

Blocking operations

- The operations/OS calls which may change the thread state to Blocked/waiting:
  - osThreadJoin
  - osThreadFlagsWait, osEventFlagsWait
  - osDelay, osDelayUntil
  - osMutexAcquire, osSemaphoreAcquire
  - osMessageQueueGet, osMessageQueuePut
- When calling from a thread, waiting time limit may be specified
  - if not needed, use osWaitForever constant
- When calling from ISR, time limit must be 0

System heap

- Memory area used for allocation of all the RTOS objects and application dynamic data
  - thread list for each priority, stacks, queues, software timers, etc.
- Heap management – heap_x.c modules
  - few different versions available, selected by user
- System must have a heap of a proper size
- To avoid hard-to-detect errors related to heap allocation, allocate objects statically
  - use static allocation options for FreeRTOS threads, timers, queues

System start

- RTOS is started by a sequence of C procedure calls in main() function
- After initializing hardware modules, create all the needed system objects – tasks, queues, timers
- FreeRTOS started by vTaskStartScheduler()/osKernelStart()
  - The function does not return

Time for real work...

- Tools used
- FreeRTOS in STM32CubeIDE

Workshop environment

- A Nucleo-144 series development board or Nucleo-64 with expansion board
- STM32CubeIDE
  - Free ST Microelectronics IDE for STM32 microcontrollers, no memory size limit, may be used for commercial projects
  - Based of ST CubeMX (application generator)

    and Atollic TrueSTUDIO (Eclipse + GNU ARM CC IDE)
  - Allows for program development and debugging

STM32CubeIDE

- STM32 application skeleton code generator
- contains µC peripheral block configurator – CubeMX
- includes middleware components, like
  - USB stack
  - FatFS
  - FreeRTOS
  - TCP/IP stack
- FreeRTOS API calls using CMSIS interface – we will use v2 → CMSIS-RTOS2

FreeRTOS in CubeIDE

- Interactive system configuration and object creation:
  - threads
  - queues and semaphores
  - timers

Thread creation

- Tasks are usually created before starting the system – xTaskCreate()/osThreadCreate()

Delays – waiting in threads

- FreeRTOS counts system timer periods and uses the software time counter value for delaying the threads
- osDelay() blocks a thread for a specified time (in ms)
  - The thread activation may be delayed because of higher priority threads
- osDelayUntil() blocks a thread until a specified time
  - osKernelGetTickCount() allows for reading the system time which may be used for osDelayUntil() argument calculation
  - To achieve fixed thread activation period:
    - Get the current system time once before the thread’s loop, using osKernelGetTickCount()
    - In a loop, calculate the argument for osDelayUntil() by adding the period to the saved time value (do not call osKernelGetTickCount() again!)

Software timers

- Optional, turned on by a FreeRTOS config option (on by default in CubeIDE).
- Turning on the timers results in the creation of a timer thread of a priority set by the programmer
    - Consider the proper priority setting
- Timer thread invokes used-defined callback functions once or periodically
- The timer callback function should not make blocking calls, as this would block the timer thread

Operations on timers

- timers are stopped by default
- Timer operations:
  - Start - osTimerStart()
  - Stop
  - Restart
- In practice, it is enough to start a periodic timer once

Idle thread

- If there is no other ready task, the system activates the idle task, of the lowest priority among the other tasks
- A programmer may define a function to be called from an idle thread void vApplicationIdleHook(void)
- The function may be used to put a processor into an energy saving (sleep) state

Thread sync and communication mechanisms

- Queues, also used as semaphores
- Usually created before system start
- A queue is characterized by the size of an element and no. of elements

Queue operations

- xQueueSendToBack()/osMessageQueuePut(), xQueueSendToFront() – insert ana object at the end or start of a quee
- xQueueReceive()/osMessageQueueGet() – get an element, waiting if necessary
  - The last call argument sets the time limit for a blocking call – when the limit is reached, the operation fails
  - osWaitForever → no time limit

Semaphores

- In FreeRTOS:
  - semaphores are implemented as queues with empty messages
  - semaphore operations are macros wrapping queue operations
- xSemaphoreGive()/osSemaphoreRelease() - semaphore up
  - Non-blockig; if the semaphore already has the highest value, the value doesn’t change
- xSemaphoreTake()/osSemaphoreAcquire() – semaphore down
  - Blocking call – not to be used from ISR

Queue sets

- Queue set allows the thread to wait for one of few possible events
- Semaphores are queues, so the may be members of queue sets
- Thread is unblocked by occurrence of any of the events from the set

FreeRTOS and interrupts

- FreeRTOS doesn’t know about interrupts
- ARM Cortex-M supports two execution levels, referred to as thread and exception in ARM architecture documents
- FreeRTOS tasks and system services run at thread level
- Scheduler is implemented as exception handler, running at the lowest exception priority level
- FreeRTOS uses three ARM Cortex-M exceptions:
  - SysTick timer interrupt – for periodic task switching
  - PendSV software-triggered interrupt – for scheduler invocation
  - SVC trap – for starting the first task only

ARM processor priority

- ARM processors support multiple hardware priority levels, not related to RTOS task priority levels
- While executing thread (application) code, the processor usually works with the lowest hardware priority
- Interrupts may have different priorities
  - more interrupt priority levels used = more synchronization problems
- Interrupt priority is always higher than base thread priority
  - Interrupts are always accepted during execution of thread application code

Preemption – RTOS and ARM

- Both preemption mechanisms are independent
  - task preemption → OS kernel software
  - exception preemption → Cortex core hardware
- RTOS cannot see the hardware preemption
- Software (RTOS) may disable interrupts if needed to block hardware preemption
- When interrupts are disabled, both hardware and OS preemption becomes impossible
  - system timer interrupt is also disabled

Disabling the interrupts

- Critical kernel operations must be non-interruptable
- During these operations, all or some (lower priority) interrupts must be disabled
- In ARM Cortex-M (unlike in most of other architectures) it is generally not necessary to turn off interrupts for critical kernel routines

Interrupts and FreeRTOS services

- ISRs may need to use system services
  - Semaphores, queues
- These services contain critical sections, guaranteeing exclusivity
- In native FreeRTOS, there is a distinct set of ISR-callable services, with names ending with “FromISR”
  - Not relevant to CMSIS-RTOS interface

Interrupt service

- Interrupts are serviced during thread’s time slice
  - It consumes the thread time
- With ARM architecture, interrupt services does not use thread stack
  - there is no need to account for interrupts when setting the thread stack size

Calling system services from ISRs

- To ensure atomicity, system services are executed with raised processor priority
  - Priority is modified by RTOS kernel software, inside system calls
  - The services' priority level is configured in FreeRTOS settings
  - FreeRTOS running on ARM does not disable the interrupts completely
- A service may be called from an interrupt of priority lower than or equal to the system service priority
  - The programmer must to set the priorities of ISRs and system calls appropriately!
- Waiting time argument must be 0 in blocking calls when a service is called from an ISR

Forced thread scheduling

- A thread may yield its time slice to the task scheduler, which may activate another ready thread immediately
  - osThreadYield()
- Yield may be especially useful in ISRs, executed within the current thread context – to force scheduling after possible change of state of some task waiting for the interrupt to occur
  - CMSIS-RTOS does this automatically in potentially unblocking calls issued from ISRs

Demo: queues, semaphore, interrupt

- Processing task:
  - gets data from receive queue, processes it, puts results into send queue
  - isolated from hardware – interacts with queues only
- Transmit task:
  - gets data from send queue
  - turns on UART Tx interrupt
  - waits for Tx semaphore
  - writes Tx data to UART

Demo: queues, semaphore, interrupt

- UART interrupt:
  - Rx: puts data into receive queue
  - Tx: turns off Tx interrupt, raises the Tx semaphore
- Tx interrupt must be kept disabled in UART when there is nothing to transmit, otherwise the interrupt will be permanently signaled

Suspending the scheduler – disabling task switching

- Temporary
  - taskENTER_CRITICAL() taskEXIT_CRITICAL()
  - disables interrupts – limited use, only for a very short time
- Locking the active task – current task stays active, interrupts are serviced
  - osKernelLock(), osKernelUnlock()

RTOS programming techniques

- A thread should only wait for events signaled via RTOS using its synchronization mechanisms.
- A thread SHOULD NOT contain a loop actively waiting for (polling) the event, not containing any blocking call
  - Such a loop, if needed, may contain osDelay()
- Blocking calls from MCU vendor-supplied libraries (like STM32 HAL) contain embedded polling loops
  - Such blocking calls should not be used in RTOS-based software design
  - When working with HAL, use non blocking calls
    - In STM32 HAL, non-blocking calls’ names end with _IT or _DMA

Interacting with non-blocking HAL calls

- A non-blocking call returns immediately, while the I/O operation is still in progress
- A thread must know that the previous I/O operation was completed
  - To use the results of an input operation
  - To start another input or output operation
- This may be achieved in two ways
  - By using the HAL callback on operation completed
    - User routine called from HAL ISR – should release the semaphore, set the thread flag or put an element into a queue
  - By ensuring that the previous operation completes – proper delays between successive operations

RTOS vs. HAL and standard library I/O

- HAL and C standard library calls are generally non-reentrant
  - New versions of stdio library may optionally be reentrant – check in your environment documentation
  - STM32 HAL functions are guarded by non-working reentry protection
- Non-reentrant functions cannot be called simultaneously from >1 thread
  - Use mutexes or designate a task for I/O, exchanging the data with another tasks via queues

Polling in RTOS threads

- For event signaling, use interrupts and RTOS communication mechanisms – queues, thread flags, semaphores
- If polling cannot be avoided, it may be implemented
  - In a software timer routine
    - When event is detected, semaphore is raised or thread flag is set
  - In a thread loop – then the loop SHOULD contain either
    - OsDelay() or osDelayUntil() call or
    - OsThreadYield() call if test result is negative
- If we use RTOS, there is no continuous polling
  - Other tasks are executed, so the polling task has some inactive periods

RTOS don’ts

- Do not:
  - Use any delays other than RTOS delays (like custom delay routines based on loops or hardware timers with software-tested flags)
    - Exception: very short, microsecond delays
  - Call communication interface routines containing polling loops, especially for UART communication
  - Write hardware polling loops without osDelay() or osThreadYield()

Evaluation

https://sube.nobleprog.com/open-tef/34811