FreeRTOS and introduction to Linux embedded

Livio Tenze

ltenze@units.it

April 1, 2025

Work experiences

Outline of lectures

- STM32H7 platform description and development environment
- FreeRTOS
  - Task management
  - Queues management
  - Interrupts
  - Resources management
  - Software timer
  - Event groups
  - Notification
  - Memory management
- Simulation environment
- Real board NUCLEO-H7A3ZI-Q

Outline of lectures

- Linux embedded
  - Introduction to Linux embedded
  - Linux architecture
  - Yocto project and toolchain
- Beaglebone Black (BBB)
- SDcard + USB-serial converter

Basic knowledge

- C (C++) programming and pointers
- Versioning systems (git), compilation process (cmake, Makefile)
- Linux base commands, console (bash)
- Docker to use containers used during lessons

Recommended texts

FreeRTOS (ST)

- Richard Barry-Using the FreeRTOS Real Time Kernel - A Practical Guide - Cortex-M3 Edition
- Mastering the FreeRTOS Real Time Kernel (`https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/releases/download/V1.1.0/Mastering-the-FreeRTOS-Real-Time-Kernel.v1.1.0.pdf`)
- FreeRTOS reference manual (`https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf`)
- FreeRTOS on STM32 v2 `https://www.youtube.com/playlist?list=PLnMKNibPkDnExrAsDpjjF1PsvtoAIBquX`
- Corso STM32 di ElettronicaIn

Recommended texts

Introduction to Embedded Linux (Beaglebone black)

- Embedded Linux System with the Yocto Project
- Using Yocto Project with BeagleBone Black
- Building embedded linux systems - 2nd edition, Yaghmour et al.
- `https://bootlin.com/pub/conferences/2011/montpellier/presentation.pdf`

Exam

- Possible project on specific topic (FreeRTOS or Linux embedded, or both)
- Oral examination on subjects treated during lessons

Software facilities

- Docker image with posix simulation environment and a webserver to download posix examples and STM32CubeIDE projects (Ubuntu 22.04)
- Docker image with Yocto ready to use environment and with jumpnowtek website (a good solution if you need to customize your linux embedded system for BBB)

Part I

ST device and IDE

Board and IDE

- Board NUCLEO-H7A3ZI-Q, Nucleo-144
- Development system STMCubeIDE, based on eclipse

FreeRTOS docker container

Instead of configuring from scratch the compilation environment, I prepared a docker container where all requirements should be met:

```
docker run --name freertos -d -p 127.0.0.1:8080:80/tcp livius147/freertos:latest
```

After start, test the following link:
`http://localhost:8080/upload/`

Show simulator compilation process with cmake

Board Nucleo-144 STM32H7A3

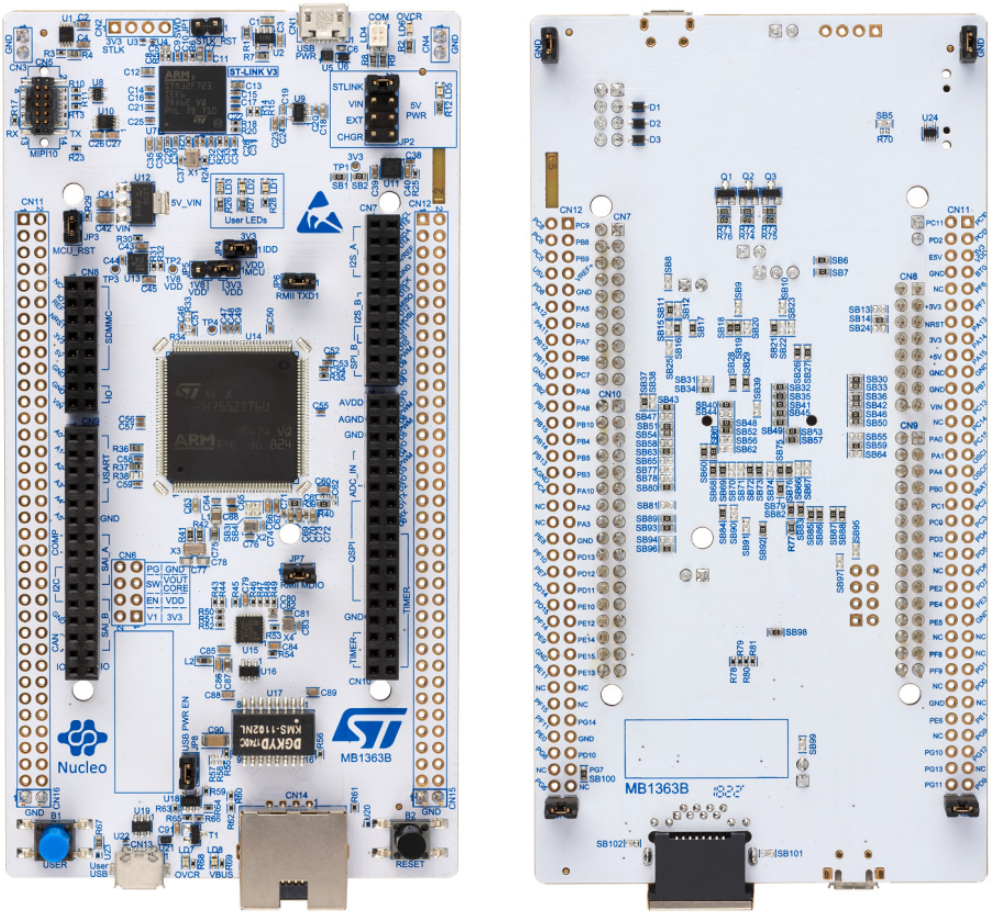


Figure: NUCLEO-H7A3ZI-Q

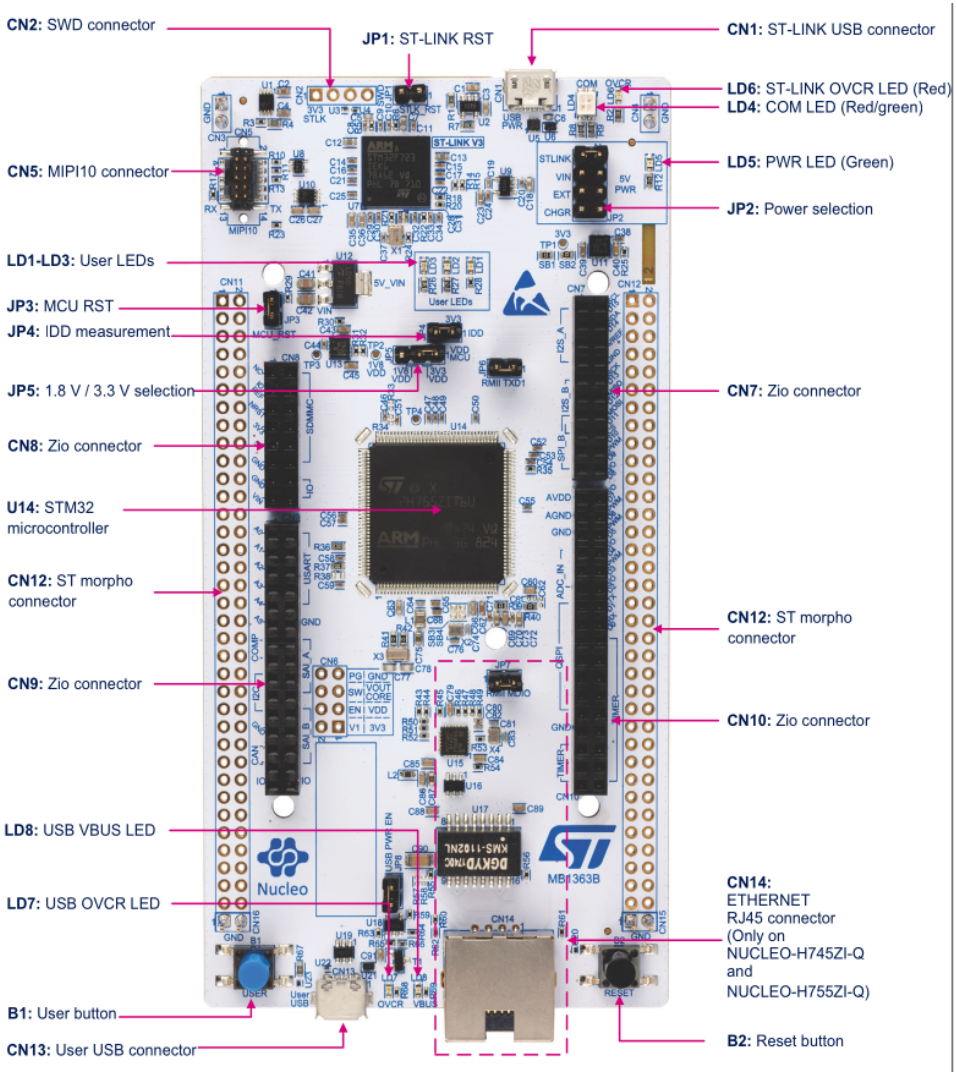
Board introduction

The STM32H7 Nucleo-144 boards based on the MB1363 reference board (NUCLEO-H745ZI-Q, NUCLEO-H755ZI-Q, **NUCLEO-H7A3ZI-Q**) provide an affordable and flexible way for users to try out new concepts and build prototypes, by choosing from the various combinations of performance and power-consumption features provided by the STM32H7microcontroller.

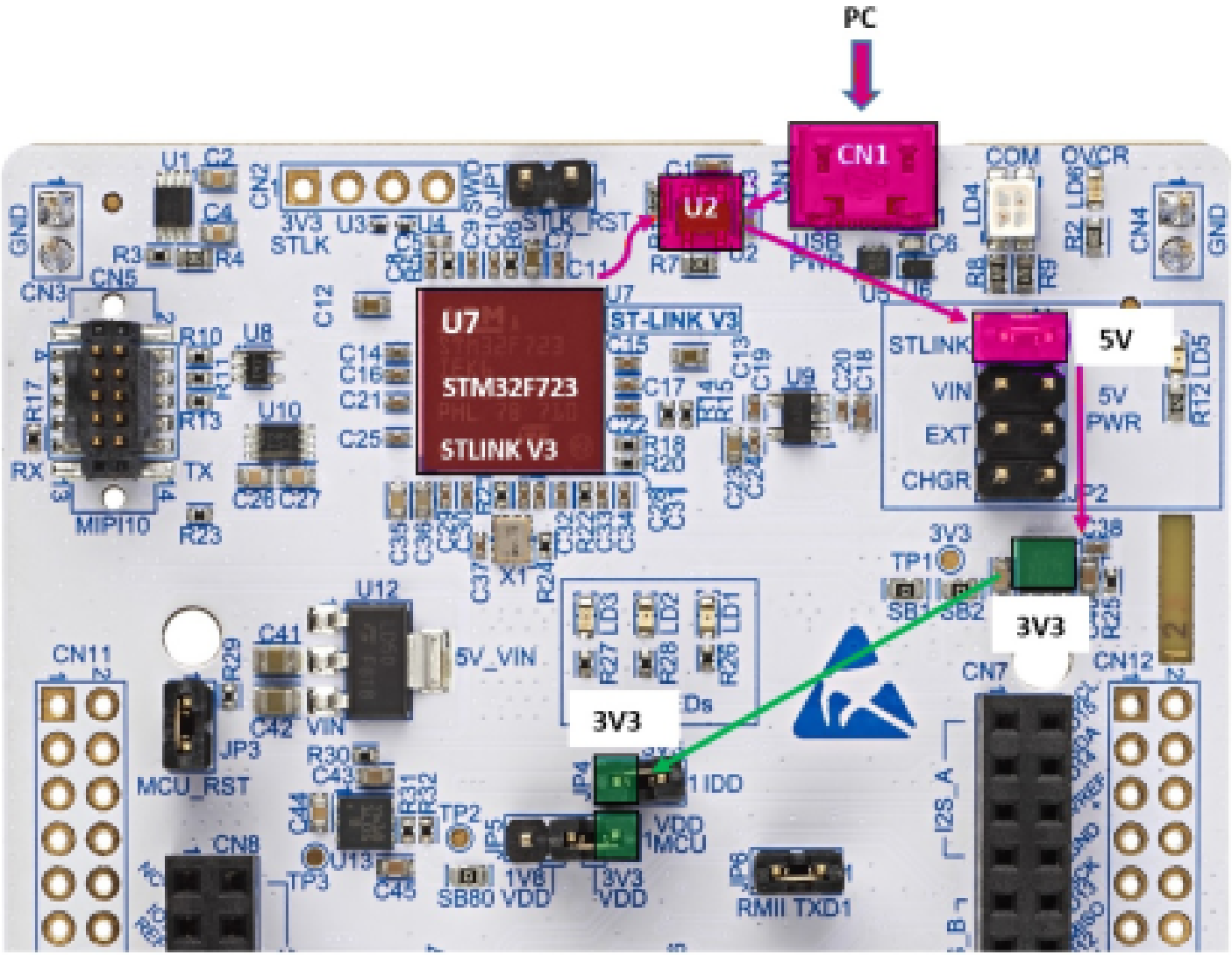
Board introduction

- The ST Zio connector, which extends the ARDUINO® Uno V3 connectivity, and the ST morpho headers provide an easy means of expanding the functionality of the Nucleo open development platform with a wide choice of specialized shields.
- The STM32H7 Nucleo-144 boards do not require any separate probe as they integrate the STLINK-V3E debugger/programmer. The STM32H7 Nucleo-144 boards come with comprehensive free software libraries and examples available with the STM32Cube MCU Package.

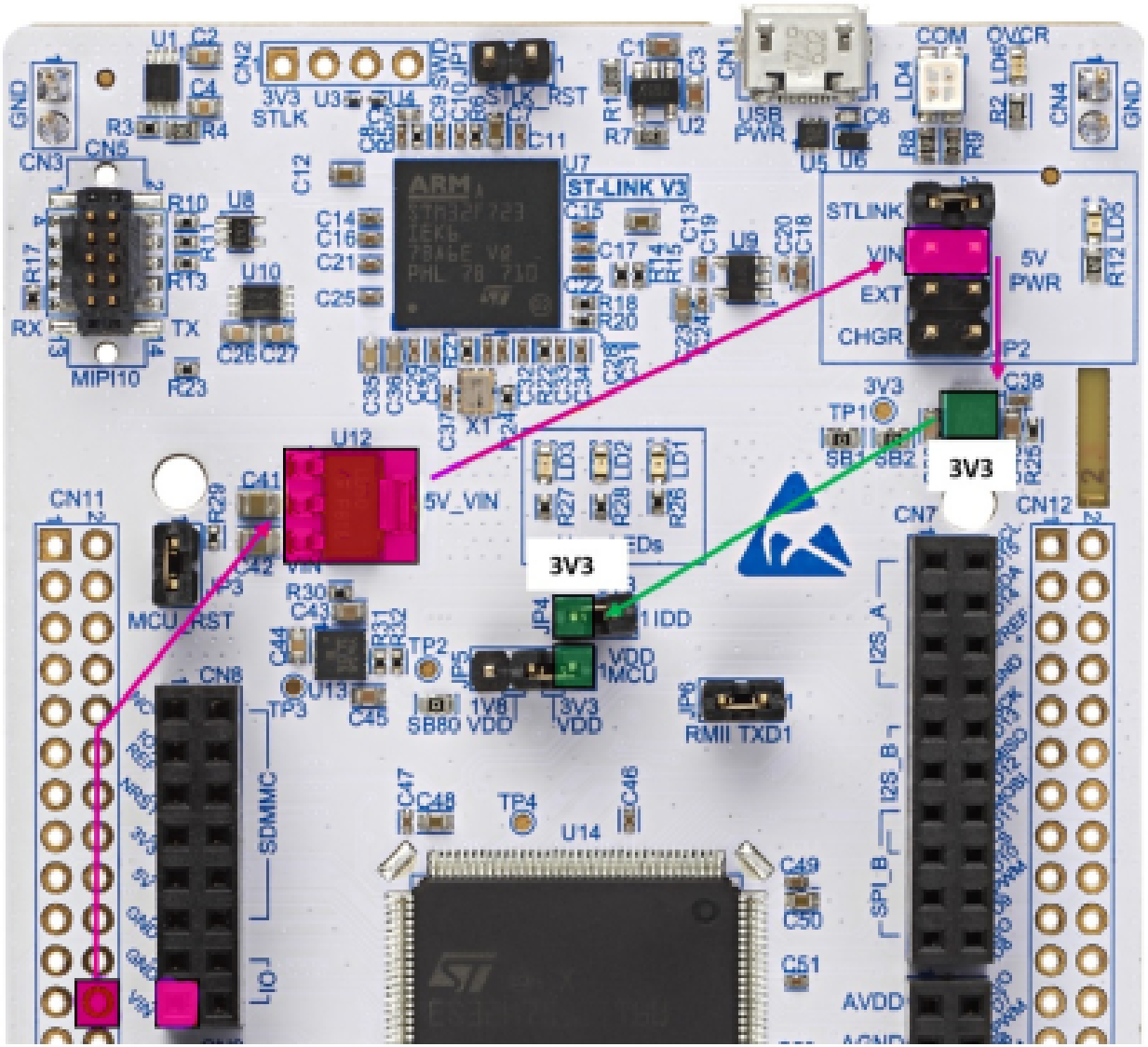
Board details



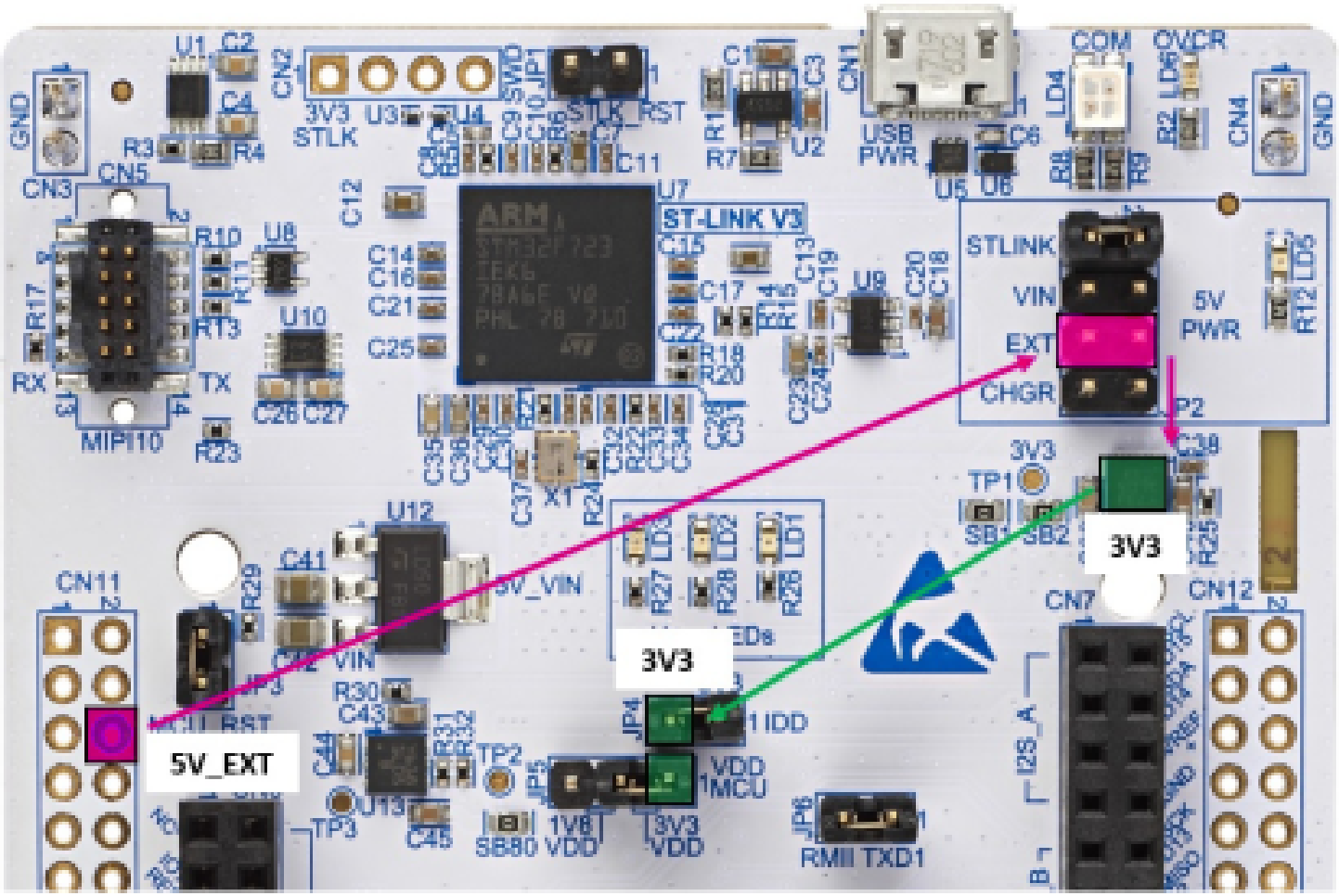
STLINK USB connector



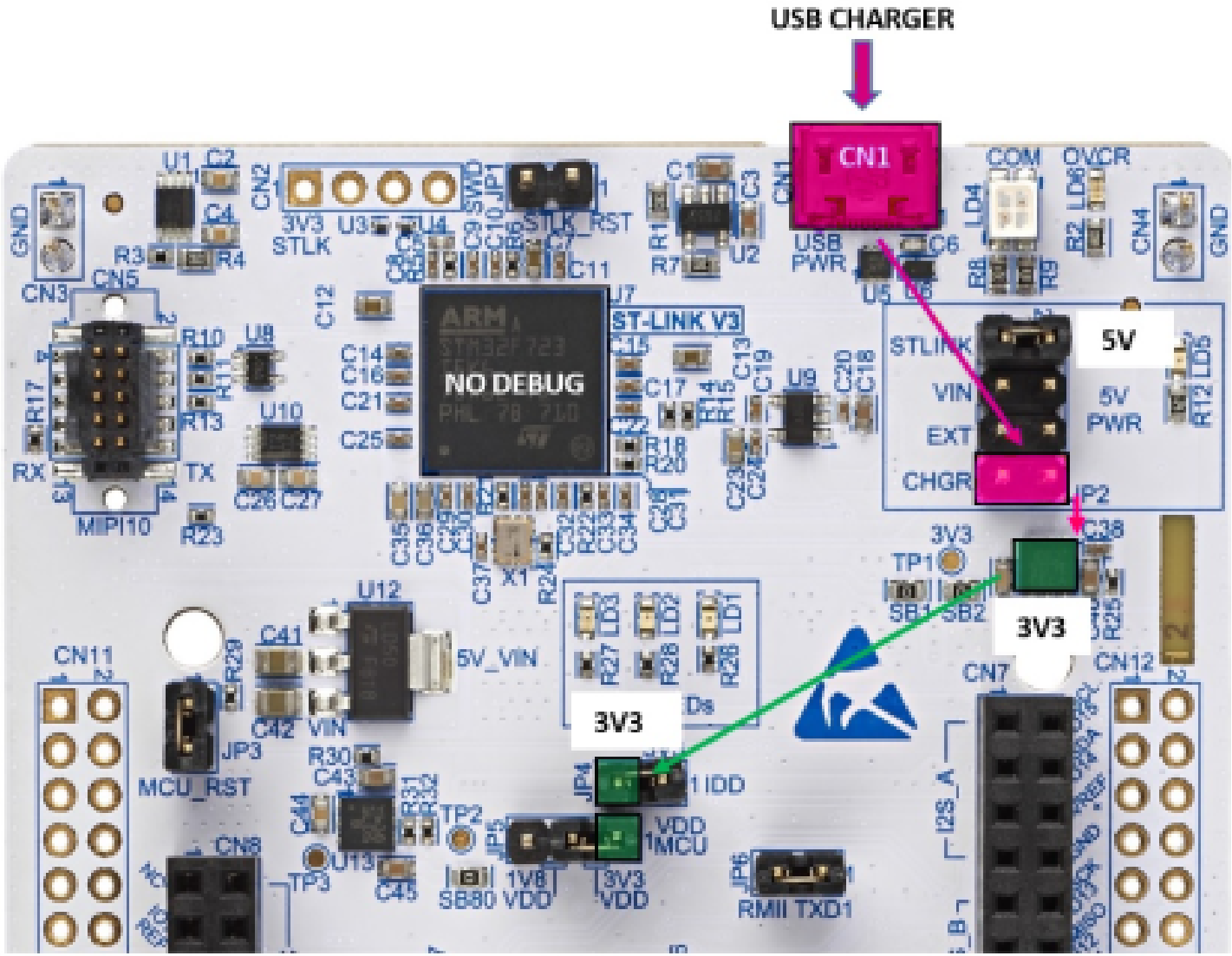
External power supply input from VIN (7-11 V, 800 mA max)



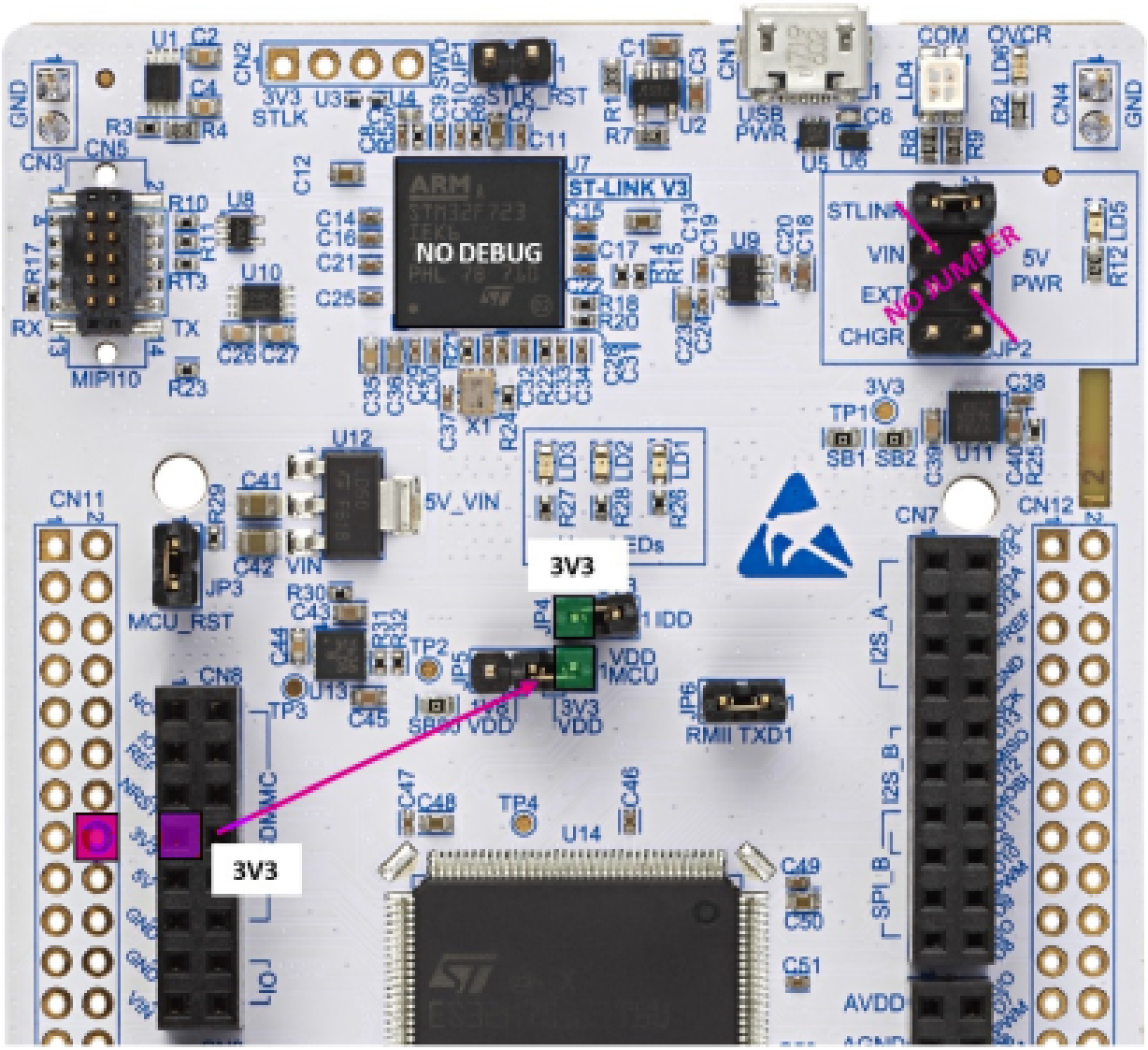
Power supply input from 5V_EXT



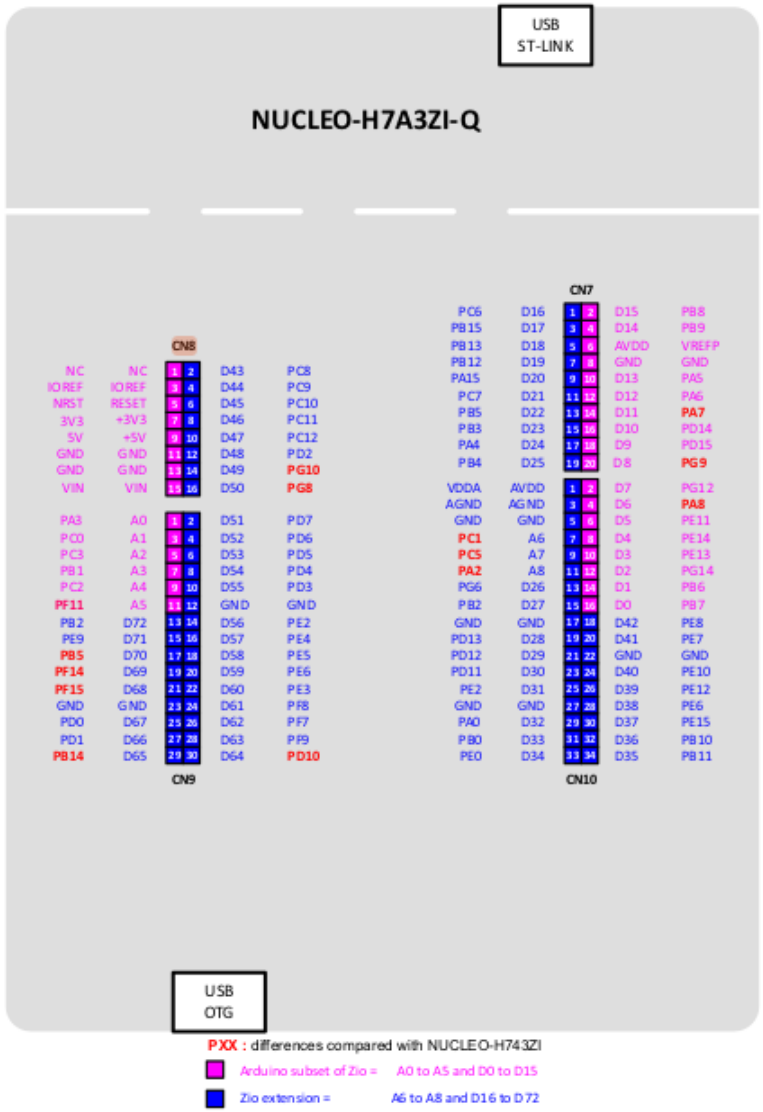
USB CHARGER (5V)



power supply input from 3V3_EXT



Extension connectors



Button, leds and other connections

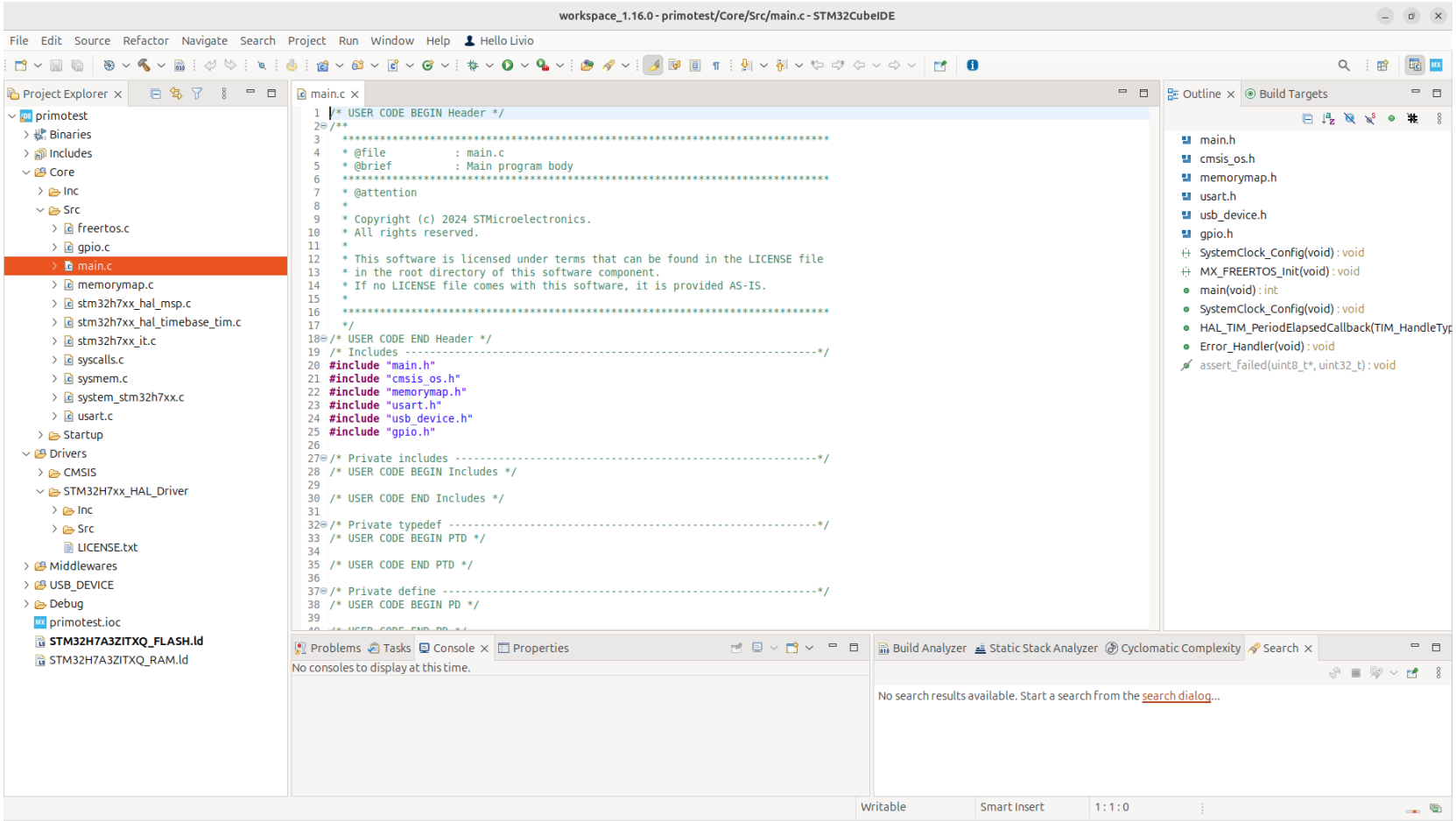
Component	GPIO
LD1 (green)	PB0
LD2 (yellow)	PE1
LD3 (red)	PB14
Button B1	PC13

UART3 and SWD are connected to STLINK and are available via the micro USB connection.

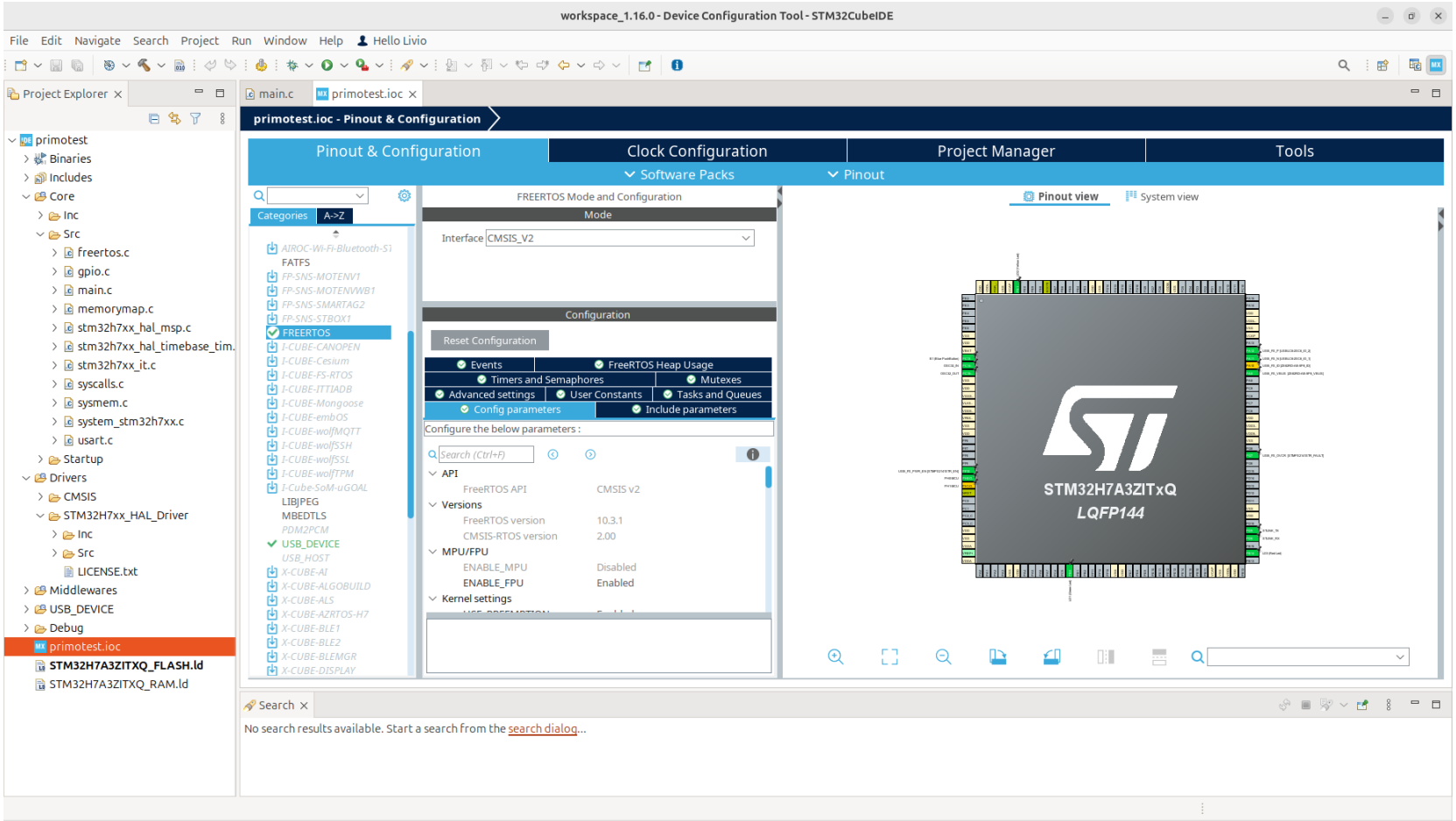
STM32H7A3ZI microcontroller

- Schematics are available to check the behaviour of the board (provided inside the freertos docker image).
- The datasheet of the microcontroller is available at `https://www.st.com/en/microcontrollers-microprocessors/stm32h7a3zi.html?rt=ds&id=DS13195`

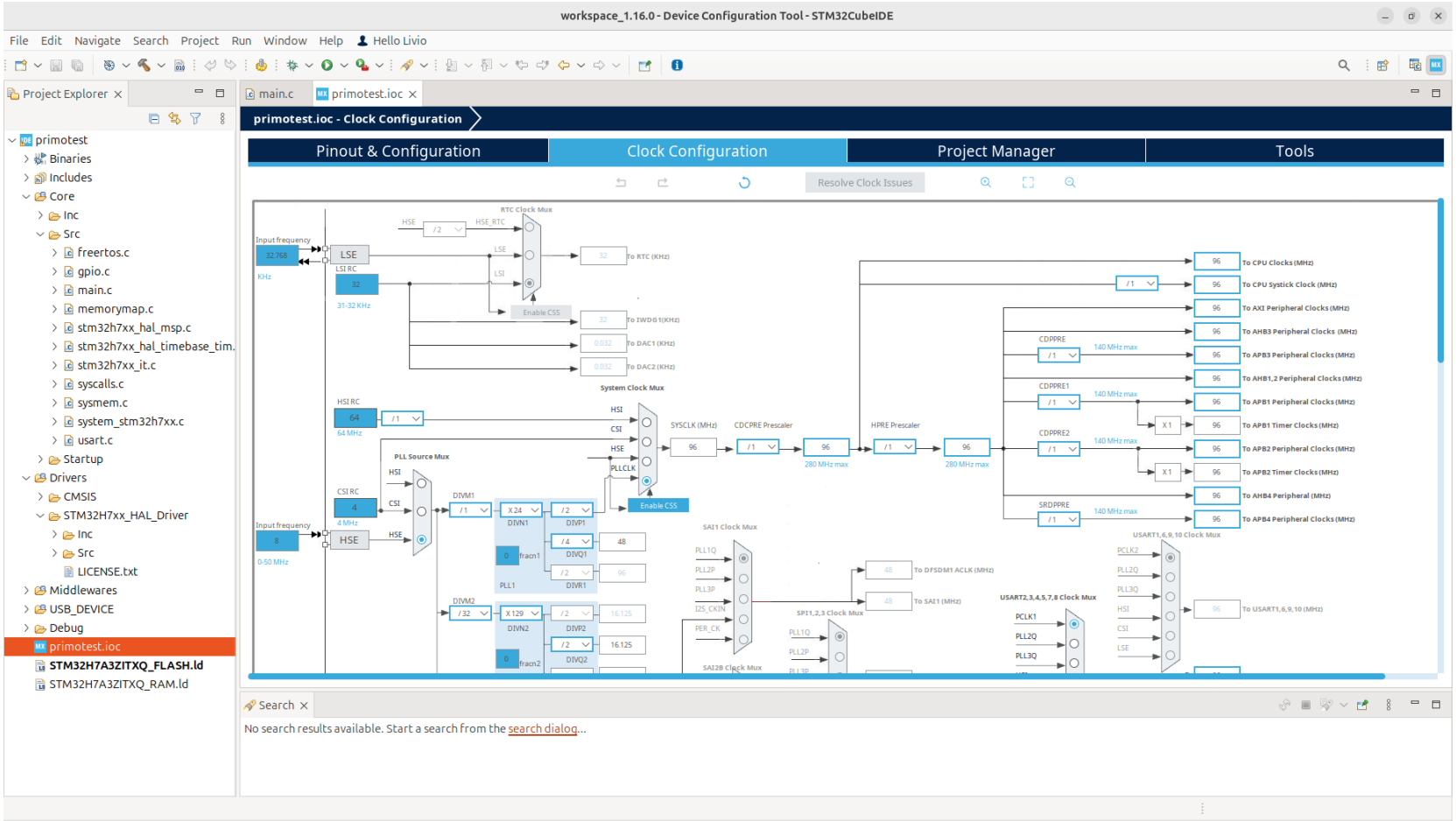
STM32CubeIDE



STM32CubeIDE FreeRTOS



STM32CubeIDE clock



Summary for IDE

- Warning: create a login account
- Show how to setup a project
- Show how to add middleware FreeRTOS
- Configure FreeRTOS and tick timer (HAL)
- Parameters and remarks about CMSIS
- Project manager $\rightarrow$ Code generator
- Blocks where to write new code, ALT+K
- Show template from docker

Part II

FreeRTOS

Multitasking on microncontroller

- Typically, applications of microncontrollers include a mix of both hard and soft real-time requirements.

Multitasking on microncontroller

- Typically, applications of microncontrollers include a mix of both hard and soft real-time requirements.
- Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless.

Multitasking on microncontroller

- Typically, applications of microncontrollers include a mix of both hard and soft real-time requirements.
- Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless.
- Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which microcontroller applications can be built to meet their hard real-time requirements.

FreeRTOS

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which microcontroller applications can be built to meet their hard real-time requirements. **It allows microcontroller applications to be organized as a collection of independent threads of execution.** As most Cortex-M3 microcontroller have only one core, in reality only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer.

In the simplest case, the application designer could assign **higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements**. This would ensure that hard real-time threads [1] are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

[1] In FreeRTOS, each thread of execution is called a task.

Why a real-time kernel

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution.

Why a real-time kernel

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution.
In more complex cases, it is likely that using a kernel would be preferable, but where **the crossover point occurs will always be subjective**.

Real-time features

- Abstracting away timing information

Real-time features

- Abstracting away timing information
  The kernel is responsible for execution timing and provides a time-related API to the application. This allows the structure of the application code to be simpler and the overall code size to be smaller.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
  Abstracting away timing details results in fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
  Tasks are independent modules, each of which should have a well-defined purpose.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
  Tasks should also have well-defined interfaces, allowing easier development by teams.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
  If tasks are well-defined independent modules with clean interfaces, they can be tested in isolation.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
  Greater modularity and fewer interdependencies can result in code that can be re-used with less effort.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
  Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Idle time

  The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Idle time
- Flexible interrupt handling
  Interrupt handlers can be kept very short by deferring most of the required processing to handler tasks.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Idle time
- Flexible interrupt handling
- Mixed processing requirements
  Simple design patterns can achieve a mix of periodic, continuous, and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities.

Real-time features

- Abstracting away timing information
- Maintainability/Extensibility
- Modularity
- Team development
- Easier testing
- Code reuse
- Improved efficiency
- Idle time
- Flexible interrupt handling
- Mixed processing requirements
- Easier control over peripherals
  Gatekeeper tasks can be used to serialize access to peripherals.

FreeRTOS features 1/2

- Pre-emptive or co-operative operation

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive mutexes

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive mutexes
- Software timers

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive mutexes
- Software timers
- Event groups

FreeRTOS features 1/2

- Pre-emptive or co-operative operation
- Optional time-slicing
- Very flexible task priority assignment
- Flexible, fast and light-weight task notification mechanisms
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive mutexes
- Software timers
- Event groups
- Stream buffer

FreeRTOS features 2/2

- Message buffers

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications (for some architectures)

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications (for some architectures)
- Memory Protection Unit support for isolating tasks and increasing application safety (for some architectures)

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications (for some architectures)
- Memory Protection Unit support for isolating tasks and increasing application safety (for some architectures)
- Software managed interrupt stack when appropriate (this can help save RAM)

FreeRTOS features 2/2

- Message buffers
- Co-routines (deprecated)
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace macros
- Task run-time statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tick-less capability for extreme low power applications (for some architectures)
- Memory Protection Unit support for isolating tasks and increasing application safety (for some architectures)
- Software managed interrupt stack when appropriate (this can help save RAM)
- The ability to create RTOS objects using either statically or dynamically allocated memory

Memory Protection Unit (MPU)

There are two versions of FreeRTOS for Cortex-M3:

- FreeRTOS-MPU includes full Memory Protection Unit (MPU) support. In this version, tasks can execute in either User mode or Privileged mode. Also, access to Flash, RAM, and peripheral memory regions can be tightly controlled, on a task-by-task basis.

Memory Protection Unit (MPU)

There are two versions of FreeRTOS for Cortex-M3:

- FreeRTOS-MPU includes full Memory Protection Unit (MPU) support. In this version, tasks can execute in either User mode or Privileged mode. Also, access to Flash, RAM, and peripheral memory regions can be tightly controlled, on a task-by-task basis.
- FreeRTOS (original) This does not include any MPU support. All tasks execute in the Privileged mode and can access the entire memory map.

Resources Used By FreeRTOS

FreeRTOS has a very small footprint. A typical kernel build will **consume approximately 6K bytes of Flash space** and **a few hundred bytes of RAM**. Each task also requires RAM to be allocated for use as the task stack.

FreeRTOS and versions

FreeRTOS MIT open source license is designed to ensure:

- FreeRTOS can be used in commercial applications.
- FreeRTOS itself remains open source.
- FreeRTOS users retain ownership of their intellectual property.

FreeRTOS and versions

- OpenRTOS is a commercially licensed version of the FreeRTOS kernel that includes indemnification and dedicated support. FreeRTOS and OPENRTOS share the same code base. OPENRTOS is provided under license from AWS by WITTENSTEIN high integrity systems - an AWS strategic partner.
- SafeRTOS has been developed in accordance with the practices, procedures, and processes necessary to claim compliance with various internationally recognized safety related standards.

License details

Data types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two port-specific data types: TickType_t and BaseType_t. The following list describes the macro or typedef used and the actual type:

- TickType_t: FreeRTOS configures a periodic interrupt called the tick interrupt. The number of tick interrupts that have occurred since the FreeRTOS application started is called the tick count. The tick count is used as a measure of time. TickType_t is the data type used to hold the tick count value, and to specify times.

Data types

- BaseType_t: this is always defined as the most efficient data type for the architecture. Typically, this is a 64-bit type on a 64-bit architecture, a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture
  It is generally used for return types that take only a very limited range of values, and for pdTRUE/pdFALSE type Booleans.

Variable names

Variables are prefixed with their type:

- 'c' for char
- 's' for int16_t (short)
- 'l' for int32_t (long)
- 'x' for BaseType_t
- 'u' for unsigned
- 'p' for a pointer

and any other non-standard types (structures, task handles, queue handles, etc.).
For example, a variable of type uint8_t will be prefixed with 'uc', and a variable of type pointer to char (char *) will be prefixed with 'pc'.

Function names

Functions are prefixed with both the type they return and the file they are defined within. For example:

- v**Task**PrioritySet() returns a void and is defined within tasks.c.
- x**Queue**Receive() returns a variable of type BaseType_t and is defined within queue.c.
- pv**Timer**GetTimerID() returns a pointer to void and is defined within timers.c.

File scope (private) functions are prefixed with '**prv**'.

Macro names

Most macros are written in upper case, and prefixed with lower case letters that indicate where the macro is defined. The following table provides a list of prefixes.

Prefix	**Location**
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Macro names

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Tasks

Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit.

Tasks

Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit. FreeRTOS tasks must not be allowed to return from their implementing function in any way they must not contain a 'return' statement and must not be allowed to execute past the end of the function. **If a task is no longer required, it should instead be explicitly deleted.**

Tasks

```
void ATaskFunction( void *pvParameters )
{
    int iVariableExample = 0;

    for( ;; )
    {
        /* The code to implement the task
        functionality will go here. */
    }
    vTaskDelete( NULL );
}
```

Some notes

- Variables can be declared just as per a normal function. Each instance of a task created using this function will have its own copy of the iVariableExample variable. This would not be true if the variable was declared static – in which case only one copy of the variable would exist and this copy would be shared by each created instance of the task.
- A task will normally be implemented as an infinite loop.
- The code to implement the task functionality will go here.
- Should the task implementation ever break out of the above loop then the task must be deleted before reaching the end of this function. The NULL parameter passed to the vTaskDelete() function indicates that the task to be deleted is the calling (this) task.

Running - Not running

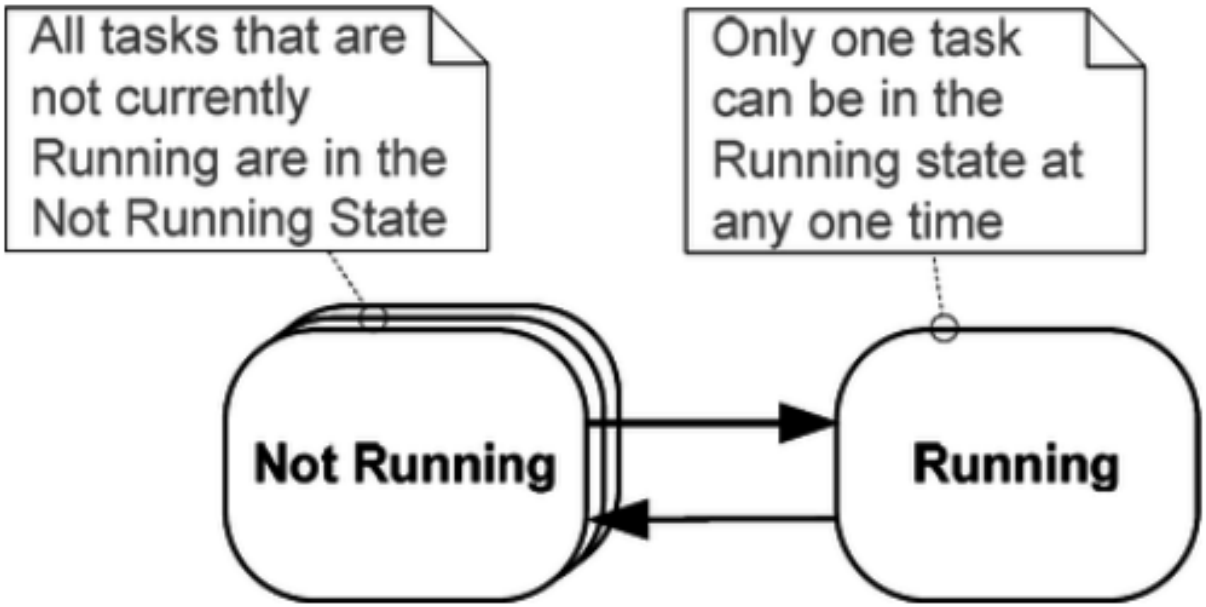
An application can consist of many tasks.

Running - Not running

An application can consist of many tasks.
If the microcontroller running the application contains a single core, then only one task can be executing at any given time. This implies that a task can exist in one of two states, Running and Not Running.

Running - Not running

An application can consist of many tasks.
If the microcontroller running the application contains a single core, then only one task can be executing at any given time. This implies that a task can exist in one of two states, Running and Not Running.



Creating Task

Tasks are created using the FreeRTOS **xTaskCreate()** API function.

```
portBASE_TYPE xTaskCreate(
      pdTASK_CODE pvTaskCode,
      const signed char *const pcName,
      unsigned short usStackDepth,
      void *pvParameters,
      unsigned portBASE_TYPE uxPriority,
      xTaskHandle *pxCreatedTask
);
```

xTaskCreate parameters

pvTaskCode The pvTaskCode parameter is simply a pointer to the function.

pcName A descriptive name for the task.

usStackDepth The usStackDepth value tells the kernel how large to make the stack. The value specifies the number of words the stack can hold, not the number of bytes. The size of the stack used by the idle task is defined by the application-defined constant configMINIMAL_STACK_SIZE, the minimum recommended for any task.

xTaskCreate parameters

pvParameters Task functions accept a parameter of type pointer to void (void*).

uxPriority Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.

pxCreatedTask pxCreatedTask can be used to pass out a handle to the task being created.

return pdTRUE This indicates that the task has been created successfully.
errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY insufficient heap memory available.

Example 1, task1

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\n";
    volatile unsigned long ul;
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++)
        {
        /* This loop is just a very crude delay implementation. There is
        nothing to do in here. Later examples will replace this crude
        loop with a proper delay/sleep function. */
        }
    }
}
```

Example 1, task2

```
void vTask2( void *pvParameters )
{
    const char *pcTaskName = "Task 2 is running\n";
    volatile unsigned long ul;
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Example 1, main

```
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(
        vTask1, /* Pointer to the function that implements the task. */
        "Task 1",/* Text name for the task. This is to facilitate
        debugging only. */
        240,
        /* Stack depth in words. */
        NULL,
        /* We are not using the task parameter. */
        /* This task will run at priority 1. */
        1,
        NULL ); /* We are not going to use the task handle. */
    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.*/
    for( ;; );
}
```

Execution pattern

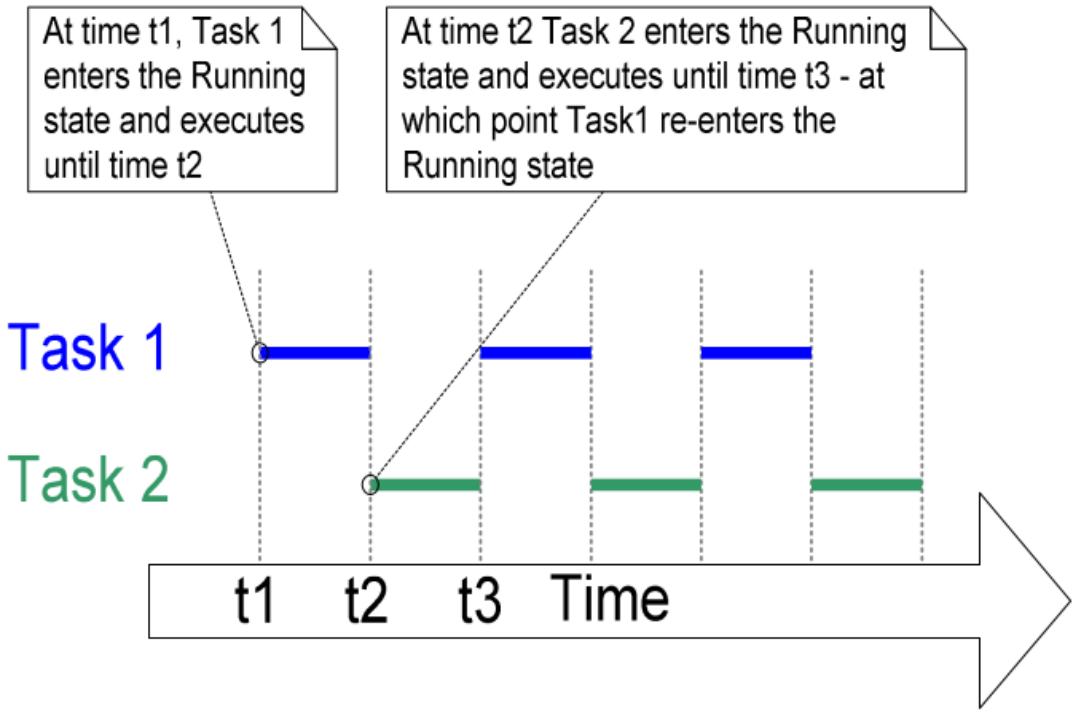


Figure: Execution pattern of example1.

Alternative task creation

Example 1 created both tasks from within main(), prior to starting the scheduler. It is also possible to create a task from within another task.

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\n";
    volatile unsigned long ul;
    /* If this task code is executing then the scheduler must already have
    been started. Create the other task before we enter the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Example 2: optimizing the code

The two tasks created in Example 1 are almost identical, the only difference between them being the text string they print out. This duplication can be removed.

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile unsigned long ul;
    /* The string to print out is passed in via the parameter.
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

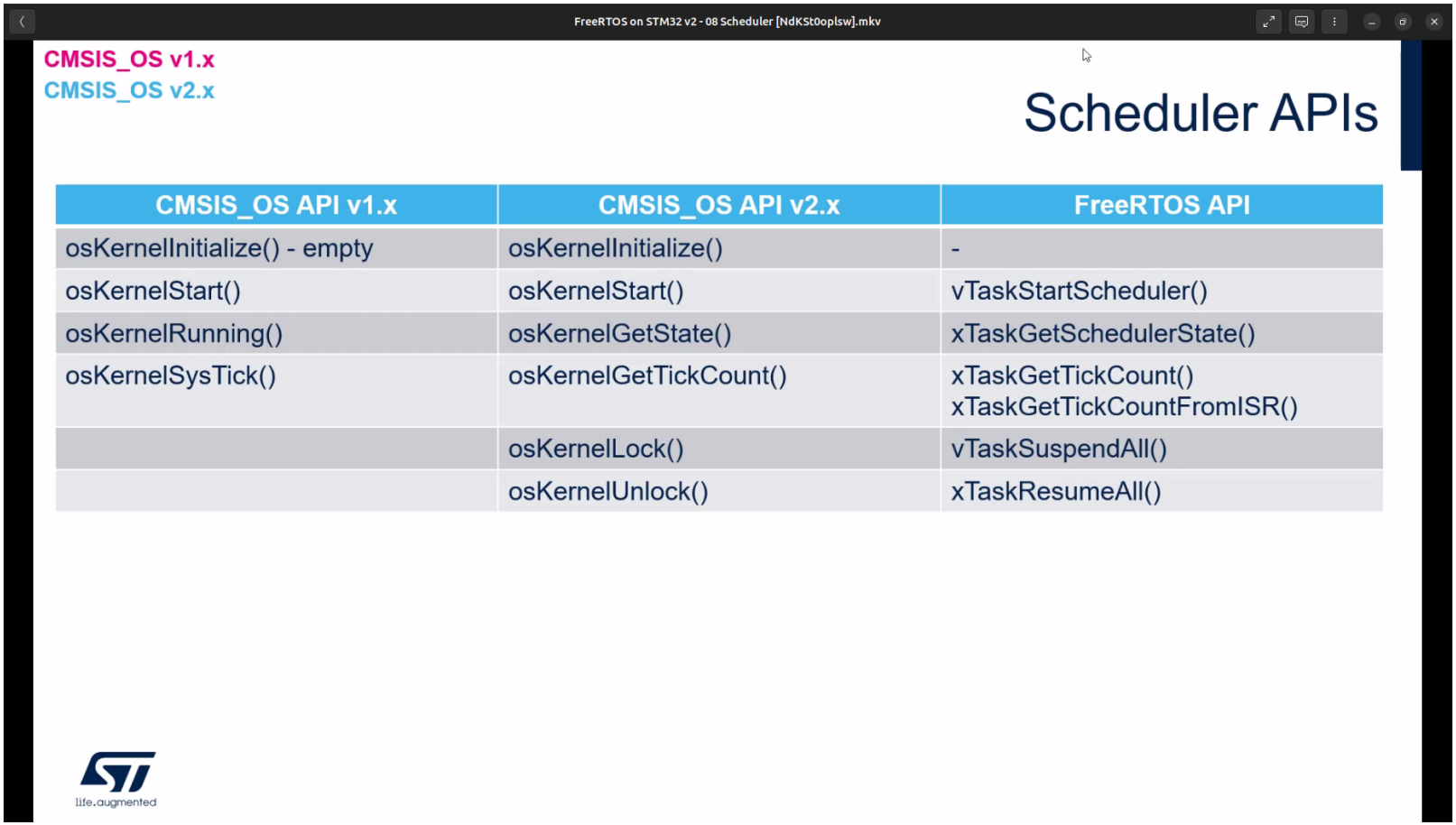
Example 2: how to use

Even though there is now only one task implementation (vTaskFunction), more than one instance of the defined task can be created. Each created instance will execute independently under the control of the FreeRTOS scheduler.

Example 2: main

```
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";
int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(     vTaskFunction,
                    "Task 1",
                    240,
                    (void*)pcTextForTask1,
                    1,
                    NULL );
    /* Create the other task in exactly the same way. Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction). Only
    the value passed in the parameter is different. Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 1, NULL );
    /* Start the scheduler so our tasks start executing. */
    vTaskStartScheduler();
    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.*/
    for( ;; );
}
```

Scheduler API in FreeRTOS ST porting



Suggested exercise

- Port the previous source code in the STMCubeIDE
- Use volatile global variables and SWV plot tracking to show the execution pattern figure
- Use vApplicationTickHook() and xTaskGetCurrentTaskHandle()

Task priorities

- The uxPriority parameter of the xTaskCreate() API function assigns an initial priority to the task being created. The priority can be changed after the scheduler has been started by using the vTaskPrioritySet() API function.
- FreeRTOS imposes no restrictions on how priorities can be assigned to tasks. Any number of tasks can share the same priority—ensuring maximum design flexibility.
- **Low numeric priority values denote low-priority tasks**, with priority 0 being the lowest priority possible.
- 0 to (configMAX_PRIORITIES – 1)

Task priorities

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn.

Each such task executes for a ‘time slice‘; it enters the Running state at the start of the time slice and exits the Running state at the end of the time slice.

Task priorities

- To be able to select the next task to run, the scheduler itself must execute at the end of each time slice. **A periodic interrupt, called the tick interrupt, is used for this purpose.** The length of the time slice is effectively set by the tick interrupt frequency, which is configured by the configTICK_RATE_HZ.
- The portTICK_RATE_MS constant is provided to allow time delays to be converted from the number of tick interrupts into milliseconds.
- The 'tick count' value is the total number of tick interrupts that have occurred since the scheduler was started; assuming the tick count has not overflowed.

Interrupt tick

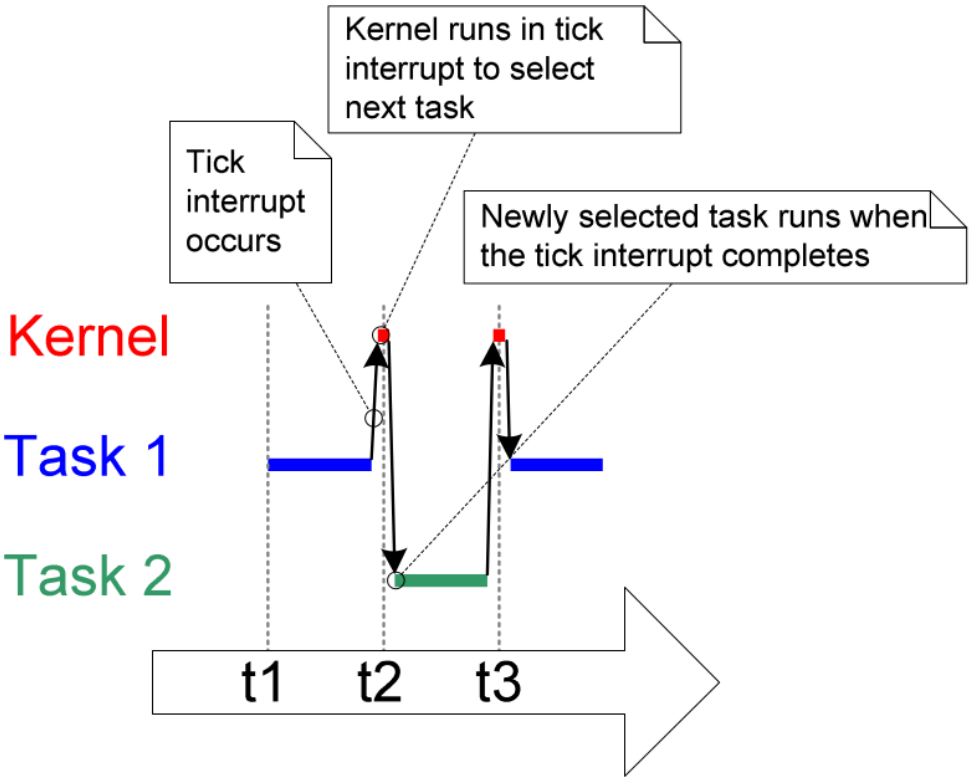
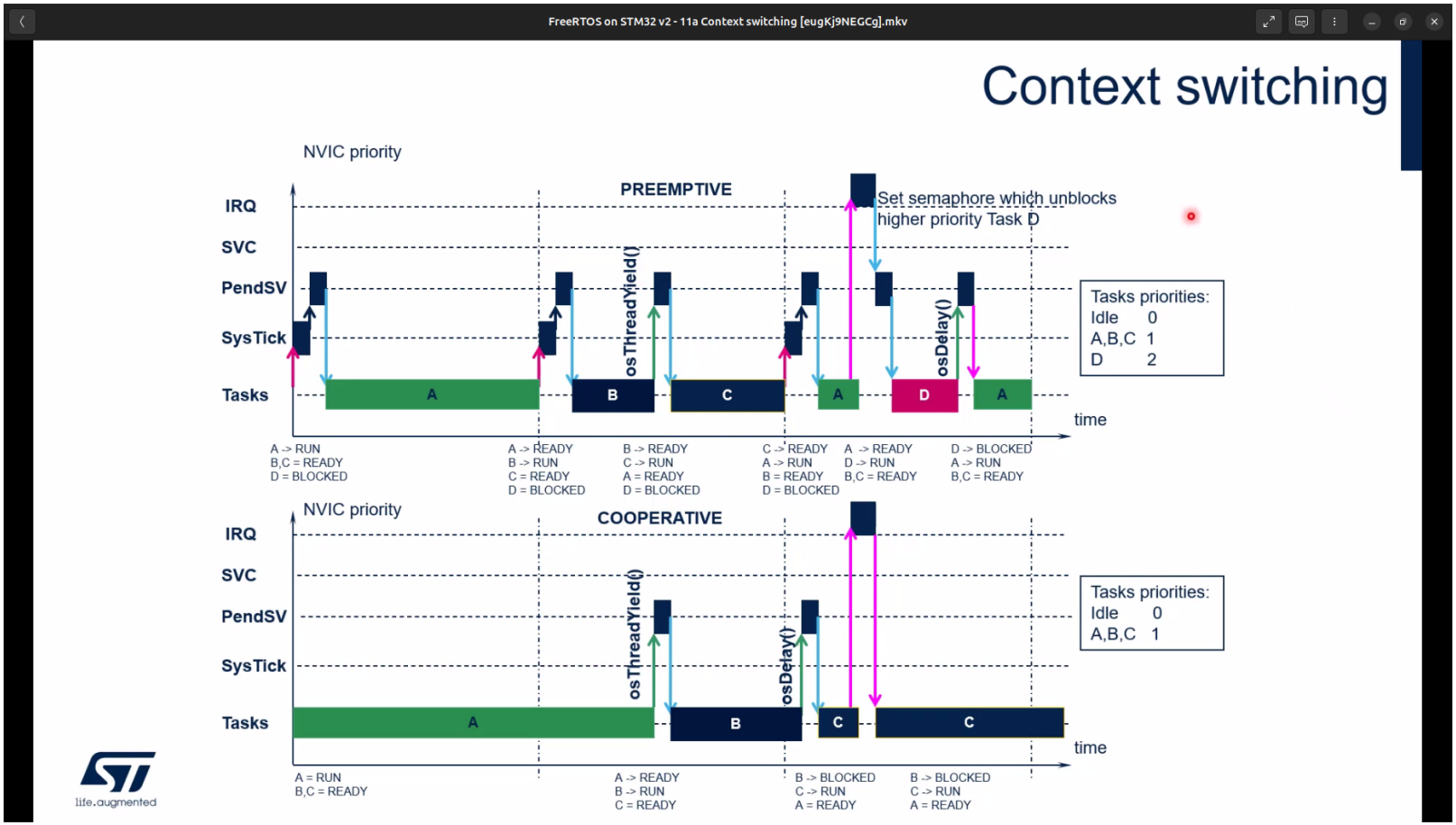


Figure: The execution sequence expanded to show the tick interrupt executing

Context switch in FreeRTOS ST porting



Example 3, change the priority

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state.

```
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";
int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );
    /* Create the second task at priority 2. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    /* If all is well we will never reach here as the scheduler will now be
    running. If we do reach here then it is likely that there was insufficient
    heap available for the idle task to be created. */
    for( ;; );
}
```

Task timing

The scheduler will always select the highest priority task that is able to run. Task 2 has a higher priority than Task 1 and is always able to run; therefore Task 2 is the only task to ever enter the Running state.

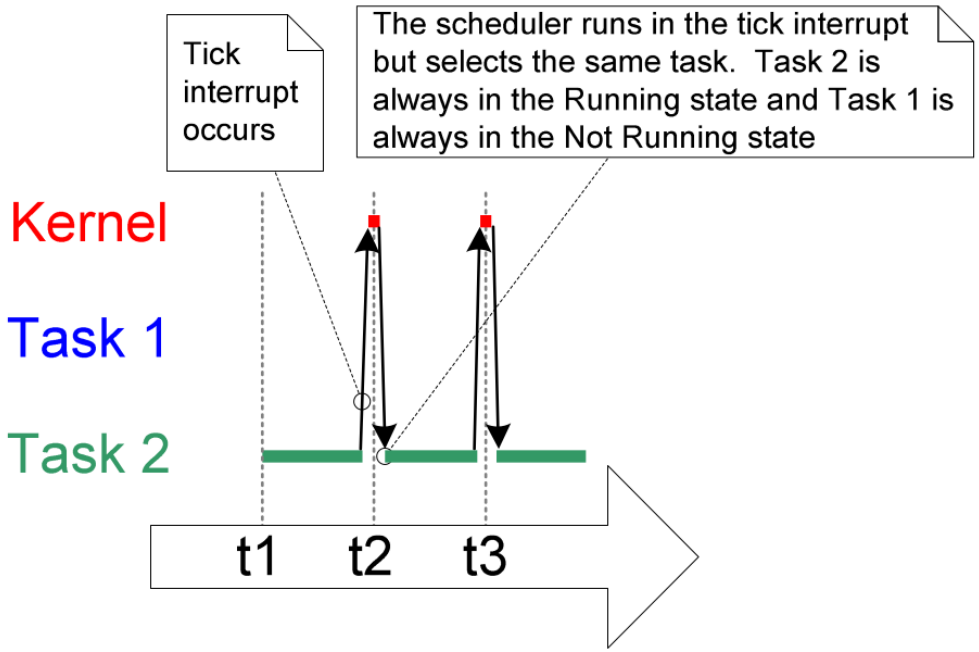


Figure: The execution pattern when one task has a higher priority than the other.

Expanding not running state

So far, the created tasks have always had processing to perform and have never had to wait for anything—as they never have to wait for anything they are always able to enter the Running state. **This type of 'continuous processing' task has limited usefulness** because they can only be created at the very lowest priority. If they run at any other priority they will prevent tasks of lower priority ever running at all.

To make our tasks useful, we need a way to allow them to be event-driven. An event-driven task has work (processing) to perform only after the occurrence of the event that triggers it, and is not able to enter the Running state before that event has occurred.

Event driven tasks

High priority tasks not being able to run means that the scheduler cannot select them and must, instead, select a lower priority task that is able to run.

Therefore, using event-driven tasks means that tasks can be created at different priorities without the highest priority tasks starving all the lower priority tasks of processing time.

Blocked state

A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.

- Temporal (time-related) events—the event being either a delay period expiring, or an absolute time being reached.
- Synchronization events—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue.

Synchronization events

FreeRTOS queues, binary semaphores, counting semaphores, recursive semaphores, and mutexes can all be used to create synchronization events.

It is possible for a task to block on a synchronization event with a timeout, effectively blocking on both types of event simultaneously.

Suspended state

Suspended is also a sub-state of Not Running.
Tasks in the Suspended state are not available to the scheduler.
The only way into the Suspended state is through a call to the **vTaskSuspend()** API function, the only way out being through a call to the **vTaskResume()** or **xTaskResumeFromISR()** API functions. Most applications do not use the Suspended state.

Ready state

Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the Ready state. They are able to run, and therefore ready to run, but are not currently in the Running state.

Transition diagram

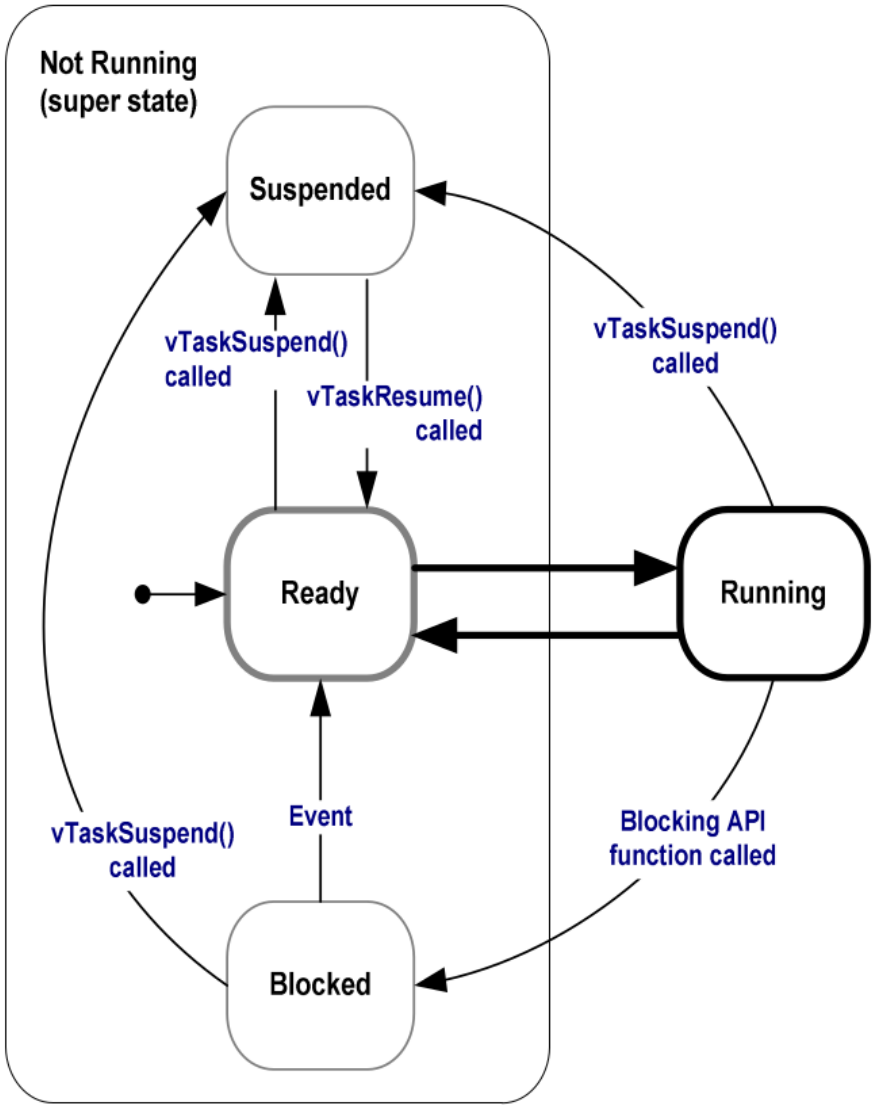


Figure: Full task state machine.

Consideration about previous examples

All the tasks created in the examples presented so far have been 'periodic'—they have delayed for a period and printed out their string, before delaying once more, and so on. The delay has been generated very crudely using a null loop—the task effectively polled an incrementing loop counter until it reached a fixed value. Example 3 clearly demonstrated the disadvantage of this method. **While executing the null loop, the task remained in the Ready state, 'starving' the other task of any processing time.** During polling, the task does not really have any work to do, but it still uses maximum processing time and so wastes processor cycles.

Solution

vTaskDelay() places the calling task into the Blocked state for a fixed number of tick interrupts. While in the Blocked state the task does not use any processing time, so processing time is consumed only when there is work to be done.

```
void vTaskDelay( portTickType xTicksToDelay );
```

xTicksToDelay The number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state.

How to use vTaskDelay()

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    /* The string to print out is passed in via the parameter.
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* Delay for a period. This time a call to vTaskDelay() is used which
        places the task into the Blocked state until the delay period has expired.
        The delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds. In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Current behaviour

Even though the two tasks are still being created at different priorities, both will now run.

The idle task is created automatically when the scheduler is started, to ensure there is always at least one task that is able to run (at least one task in the Ready state).

Example 4: timing

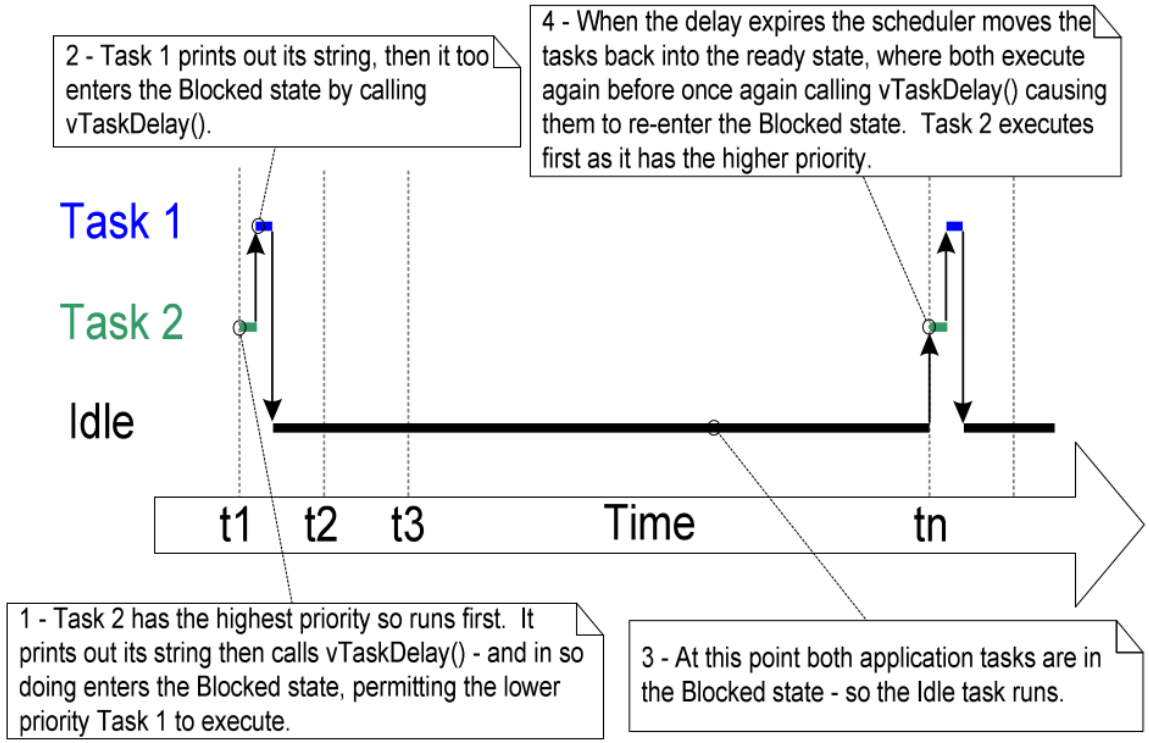


Figure: The execution sequence when the task uses vTaskDelay().

Remarks

Only the implementation of our two tasks has changed, not their functionality. Comparing previous code with the current one demonstrates clearly that **this functionality is being achieved in a much more efficient manner.**
Each time the tasks leave the Blocked state they execute for a fraction of a tick period before re-entering the Blocked state. Most of the time there are no application tasks that are able to run (no application tasks in the Ready state).
While this is the case, the idle task will run. The amount of processing time the idle task gets is **a measure of the spare processing capacity** in the system.

Status diagram

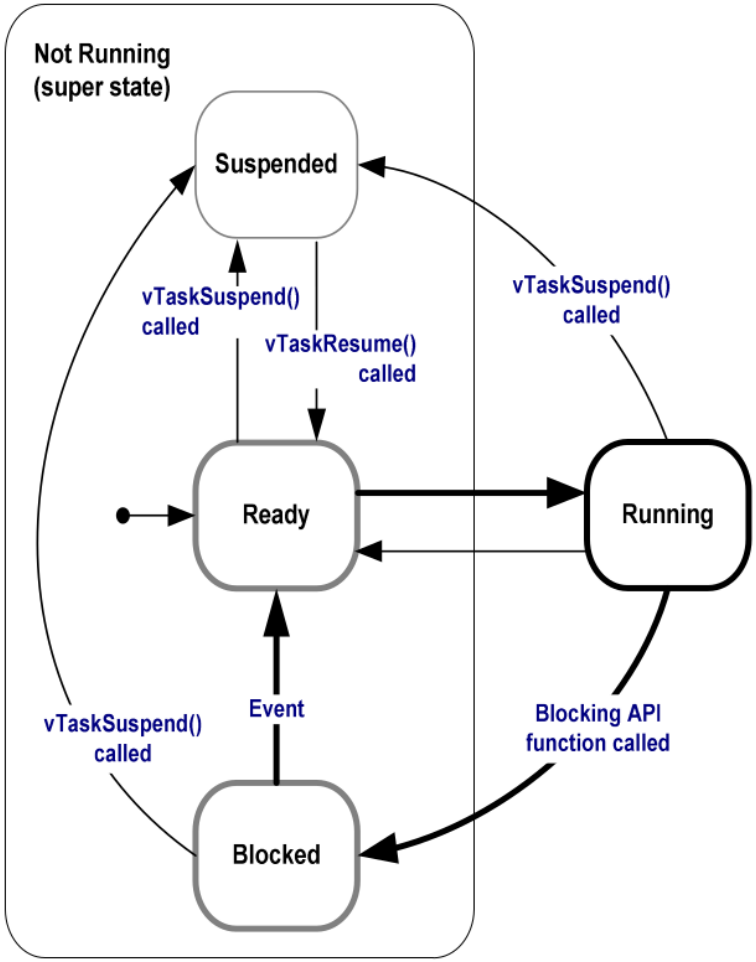


Figure: State transitions performed by the task.

vTaskDelayUntil()

vTaskDelayUntil() is similar to vTaskDelay().
As just demonstrated, the vTaskDelay() parameter specifies the number of tick interrupts that should occur between a task calling vTaskDelay() and the same task once again transitioning out of the Blocked state.

The actual time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called.

The parameters to vTaskDelayUntil() specify, instead, the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state.

vTaskDelayUntil()

```
void vTaskDelayUntil(    portTickType *pxPreviousWakeTime,
                         portTickType xTimeIncrement );
```

pxPreviousWakeTime This time is used as a reference point to calculate the time at which the task should next leave the Blocked state. The variable pointed to by pxPreviousWakeTime is updated automatically within the vTaskDelayUntil() function.

xTimeIncrement xTimeIncrement is specified in 'ticks'

Example 5: periodicity

The two tasks created in Example 4 are periodic tasks, but using vTaskDelay() does not guarantee that the frequency at which they run is fixed, as the time at which the tasks leave the Blocked state is relative to when they call vTaskDelay(). Converting the tasks to use vTaskDelayUntil() instead of vTaskDelay() solves this potential problem.

Solution

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    portTickType xLastWakeTime;
    /* The string to print out is passed in via the parameter.
    character pointer. */
    pcTaskName = ( char * ) pvParameters;
    xLastWakeTime = xTaskGetTickCount();
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );
        /* This task should execute exactly every 250 milliseconds. As per
        the vTaskDelay() function, time is measured in ticks, and the
        portTICK\_RATE\_MS constant is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil() so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

Example 6: how to mix polling and blocking tasks

This example re-enforces the stated expected system behavior by demonstrating an execution sequence when the two schemes are combined, as follows:

- Two tasks are created at priority 1. These do nothing other than **continuously** print out a string.
- The third task also just prints out a string, but this time **periodically**, so uses the vTaskDelayUntil() API function to place itself into the Blocked state between each print iteration.

Example 6: code 1

```
void vContinuousProcessingTask( void *pvParameters )
{
    char *pcTaskName;
    /* The string to print out is passed in via the parameter.
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

Example 6: code 2

```
void vPeriodicTask( void *pvParameters )
{
    portTickType xLastWakeTime;
    /* The xLastWakeTime variable needs to be initialized with the current tick
    count. Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running...\n" );
        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &LastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

Timing diagram

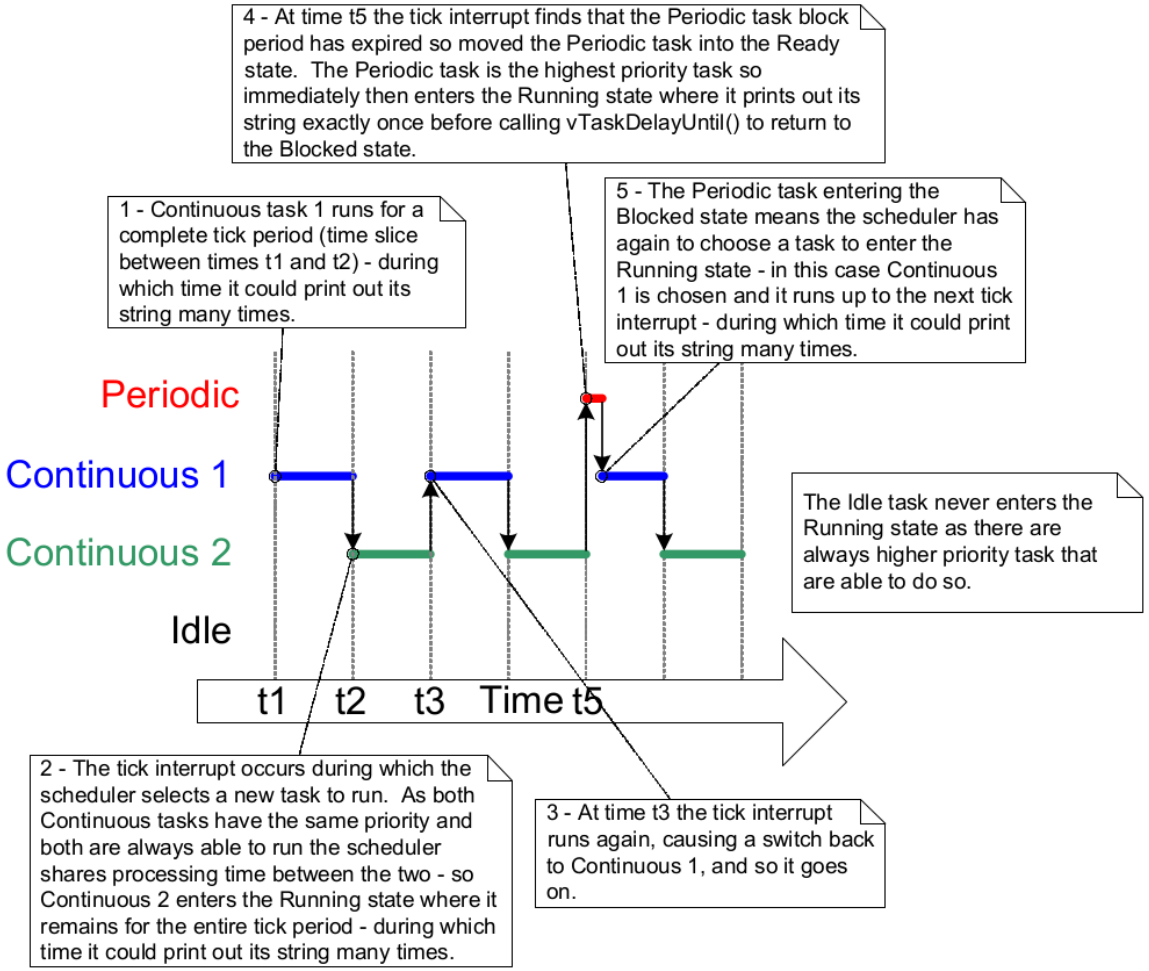


Figure: Example 6: timing diagram.

The idle task

- The tasks created in Example 4 spend most of their time in the Blocked state. While in this state, they are not able to run and cannot be selected by the scheduler.
- The processor always needs something to execute—there must always be at least one task that can enter the Running state. To ensure this is the case, an Idle task is automatically created by the scheduler when vTaskStartScheduler() is called. The idle task does very little more than sit in a loop—so, like the tasks in the original examples, it is always able to run.
- The idle task has the lowest possible priority (priority zero).

Hook function

It is possible to add application specific functionality directly into the idle task through the use of an idle hook.
Common uses:

- Executing low priority, background, or continuous processing.
- Measuring the amount of spare processing capacity.
- **Placing the processor into a low power mode.**

Limitations on the implementation of idle task

- An idle task hook function **must never attempt to block or suspend**. Blocking the idle task in any way could cause a scenario where no tasks are available to enter the Running state.
- If the application makes use of the vTaskDelete() API function then the Idle task hook must always return to its caller within a reasonable time period. This is because the Idle task is responsible for cleaning up kernel resources after a task has been deleted. **If the idle task remains permanently in the Idle hook function, then this clean-up cannot occur.**

```
void vApplicationIdleHook( void );
```

Simple idle hook

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;
/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.

...to print out the ulIdleCycleCount

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    /* The string to print out is passed in via the parameter.
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );
        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Show Example 7

Changing the task priority

The vTaskPrioritySet() API function can be used to change the priority of any task after the scheduler has been started.

Note that the vTaskPrioritySet() API function is available only when INCLUDE_vTaskPrioritySet is set to 1 in FreeRTOSConfig.h.

```
void vTaskPrioritySet(xTaskHandle pxTask,
                      unsigned portBASE_TYPE uxNewPriority);
```

Changhing task priority

pxTask The handle of the task whose priority is being modified. A task can change its own priority by passing NULL in place of a valid task handle.

uxNewPriority The priority to which the subject task is to be set. This is capped automatically to the maximum available priority of (configMAX_PRIORITIES – 1)

uxTaskPritorityGet()

The uxTaskPriorityGet() API function can be used to query the priority of a task.

The uxTaskPriorityGet() API function is available only when INCLUDE_vTaskPriorityGet is set to 1 in FreeRTOSConfig.h.

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
```

Example 8, changing task priority

The scheduler will always select the highest Ready state task as the task to enter the Running state. Example 8 demonstrates this by using the vTaskPrioritySet() API function to change the priority of two tasks relative to each other.

- Task 1 is created with the highest priority. Task 1 prints out a couple of strings before raising the priority of Task 2 to above its own priority.
- Task 2 starts to run (enters the Running state) as soon as it has the highest relative priority. Only one task can be in the Running state at any one time; so, when Task 2 is in the Running state, Task 1 is in the Ready state.

Example 8, changing task priority

- Task 2 prints out a message before setting its own priority back to below that of Task 1.
- Task 2 setting its priority back down means Task 1 is once again the highest priority task, so Task 1 re-enters the Running state.

Example 8, source code

```
void vTask1( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;
    /* This task will always run before Task 2 as it is created with the higher
    priority. Neither Task 1 nor Task 2 ever block so both will always be in either
    the Running or the Ready state.
    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );
        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks). Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet(). Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );
        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

Example 8, source code

```
void vTask2( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;
    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority. Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.
    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.
        Print out the name of this task. */
        vPrintString( "Task2 is running\n" );
        /* Set our priority back down to its original value. Passing in NULL
        as the task handle means "change my priority". Setting the
        priority below that of Task 1 will cause Task 1 to immediately start
        running again - pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Example 8, source code

```
/* Declare a variable that is used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;
int main( void )
{
    /* Create the first task at priority 2. The task parameter is not used
    and set to NULL. The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */
    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1. Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter ____^^^^^^^^^^^^^^ */
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    */
    for( ;; );
}
```

Example 8, timing diagram

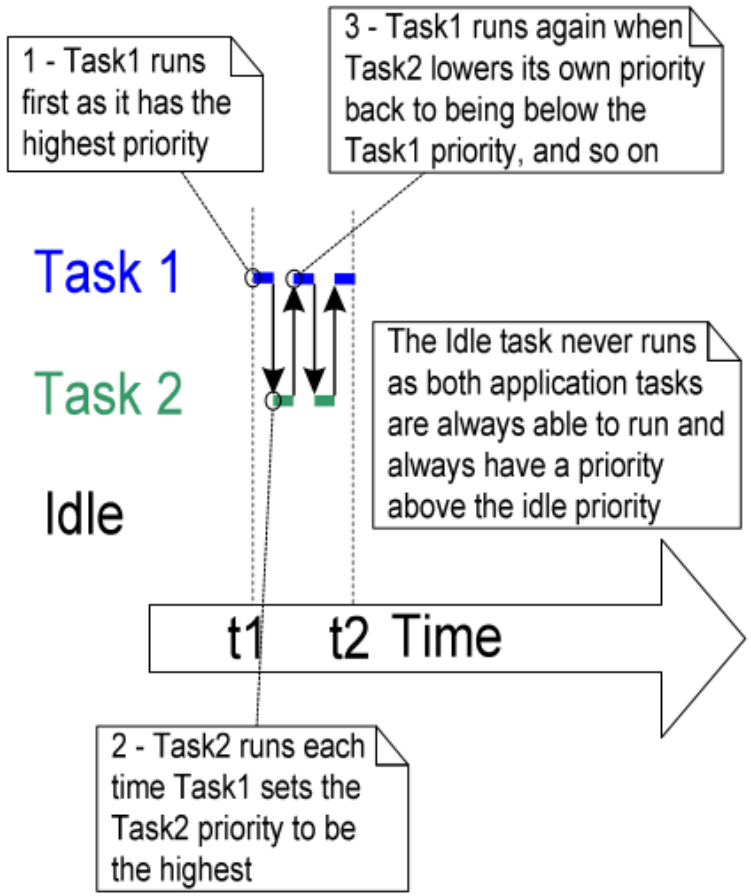


Figure: The sequence of task execution.

Deleting a Task

A task can use the vTaskDelete() API function to delete itself or any other task. Note that the vTaskDelete() API function is available only when INCLUDE_vTaskDelete is set to 1 in FreeRTOSConfig.h.

It is the responsibility of the idle task to free memory allocated to tasks that have since been deleted. Therefore, it is important that applications using the vTaskDelete() API function do not completely starve the idle task of all processing time.

Any memory or other resource that the implementation of the task allocates itself must be freed explicitly.

vTaskDelete()

```
void vTaskDelete(xTaskHandle pxTaskToDelete);
```

pxTaskToDelete The handle of the task that is to be deleted (the subject task) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.
A task can delete itself by passing NULL in place of a valid task handle.

Example 9: explanation

- Task 1 is created by main() with priority 1. When it runs, it creates Task 2 at priority 2. Task 2 is now the highest priority task, so it starts to execute immediately.
- Task 2 does nothing but delete itself.
- When Task 2 has been deleted, Task 1 is again the highest priority task, so continues executing—at which point it calls vTaskDelay() to block for a short period.
- The Idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.
- When Task 1 leaves the blocked state it again becomes the highest priority Ready state task and so pre-empts the Idle task. When it enters the Running state it creates Task 2 again, and so it goes on.

Example 9: code

```
int main( void )
{
    /* Create the first task at priority 1. The task parameter is not used
    so is set to NULL. The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 1, NULL );
    /* The task is created at priority 1 _____^. */
    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();
    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

Example 9: code

```
void vTask1( void *pvParameters )
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );
        /* Create task 2 at a higher priority. Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 240, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter ____^^^^^^^^^^^^^^ */
        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself. Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Example 9: code

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself. To do this it could call vTaskDelete()
    using NULL as the parameter, but instead and purely for demonstration purposes it
    instead calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task2 is running and about to delete itself\n" );
    vTaskDelete( xTask2Handle );
}
```

Example 9: timing diagram

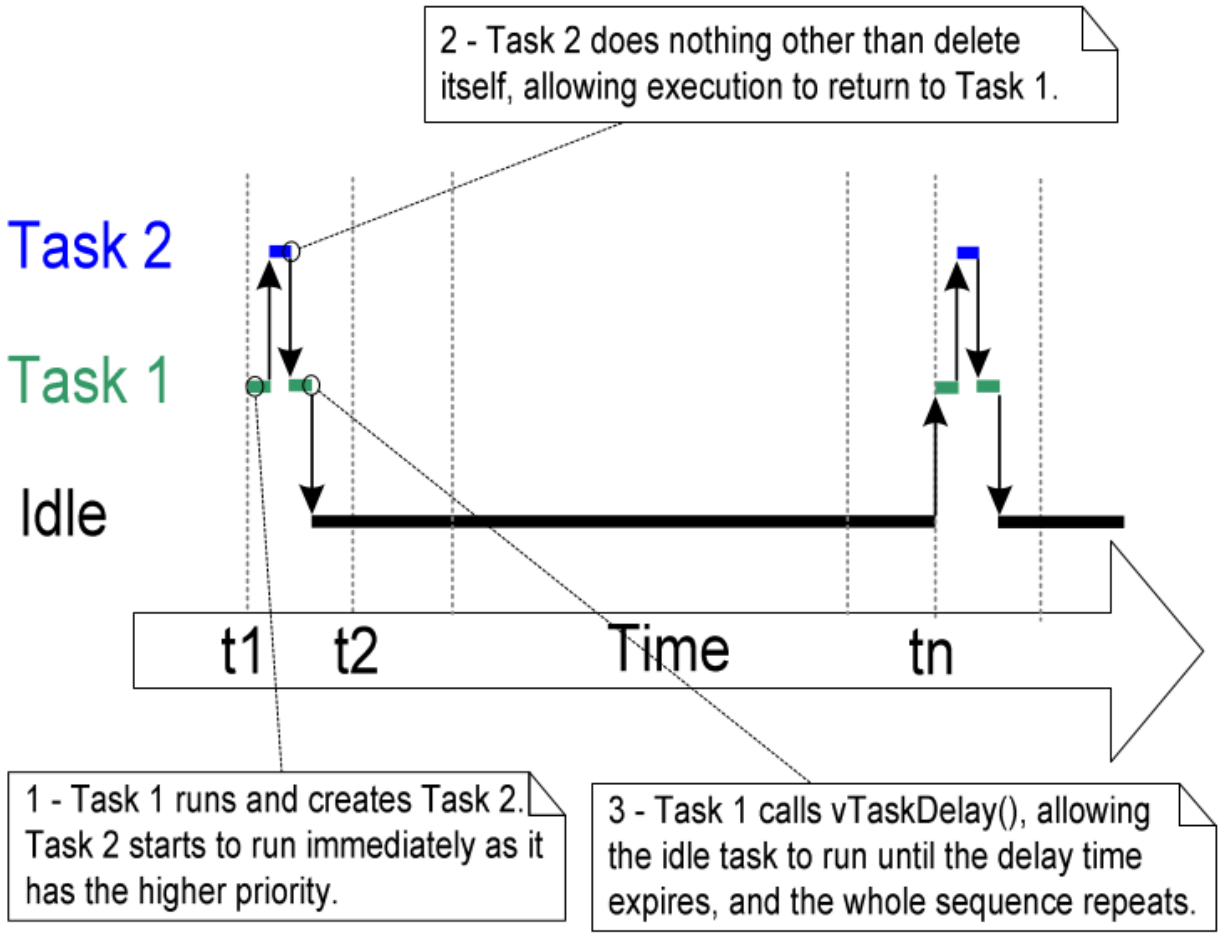


Figure: Example 9: timing diagram.

A review of scheduling algorithms

The examples illustrate how and when FreeRTOS selects which task should be in the Running state.

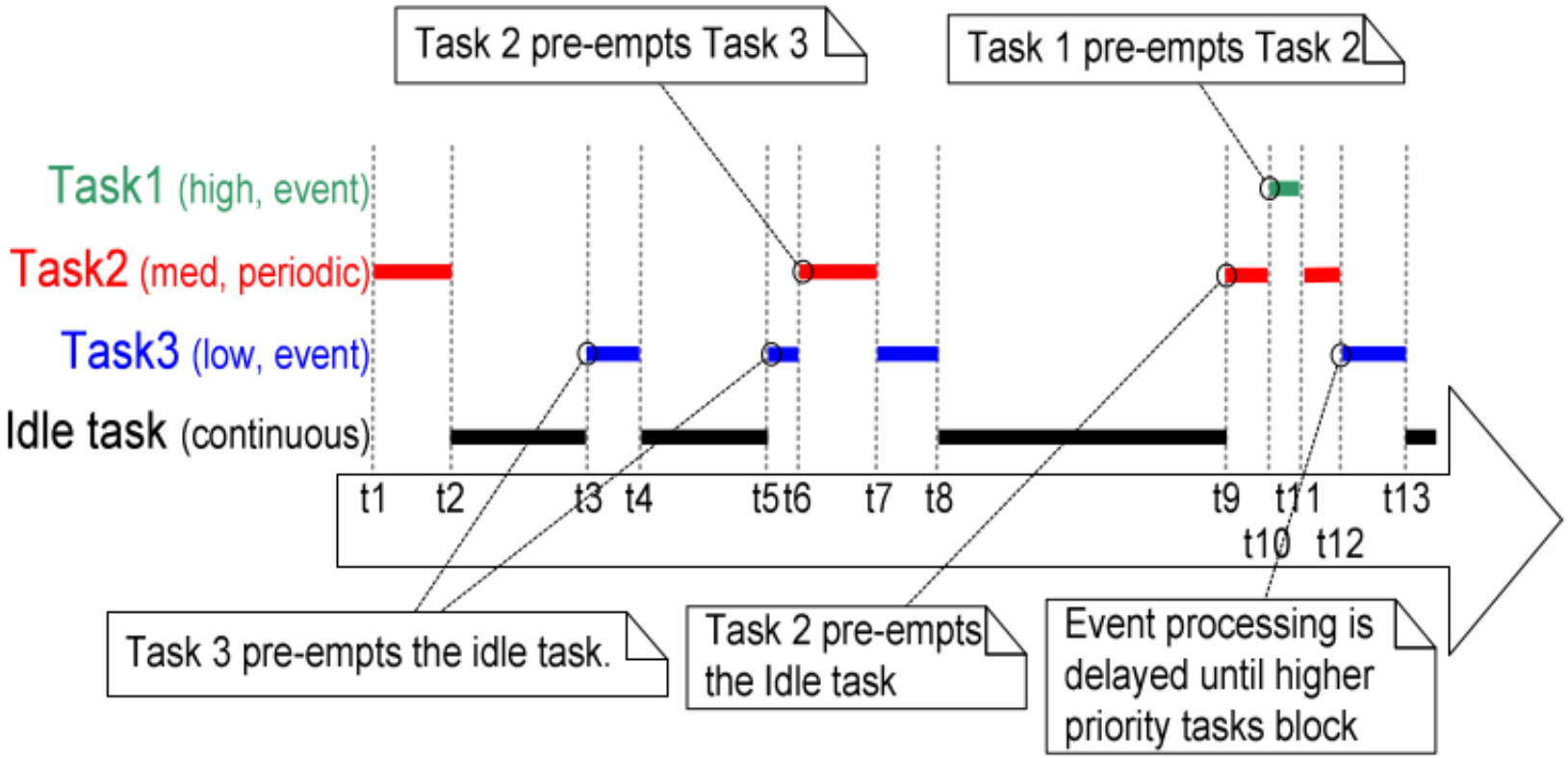
- Each task is assigned a priority.
- Each task can exist in one of several states.
- Only one task can exist in the Running state at any one time.
- The scheduler always selects the highest priority Ready state task to enter the Running state.

This type of scheme is called Fixed Priority Pre-emptive Scheduling

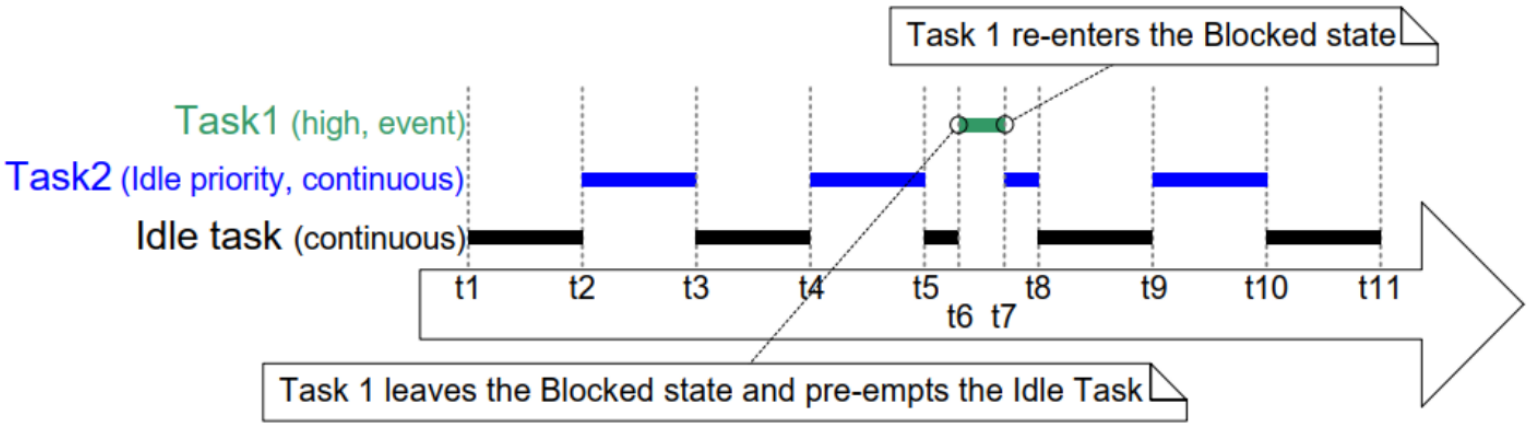
Fixed priority preemptive scheduling

Tasks can wait in the Blocked state for an event and are automatically moved back to the Ready state when the event occurs.
Temporal events occur at a particular time—for example, when a block time expires. behavior. They are generally used to implement periodic or timeout Synchronization events occur when a task or interrupt service routine sends information to a **queue** or to one of the many types of **semaphore**. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

Preemption example

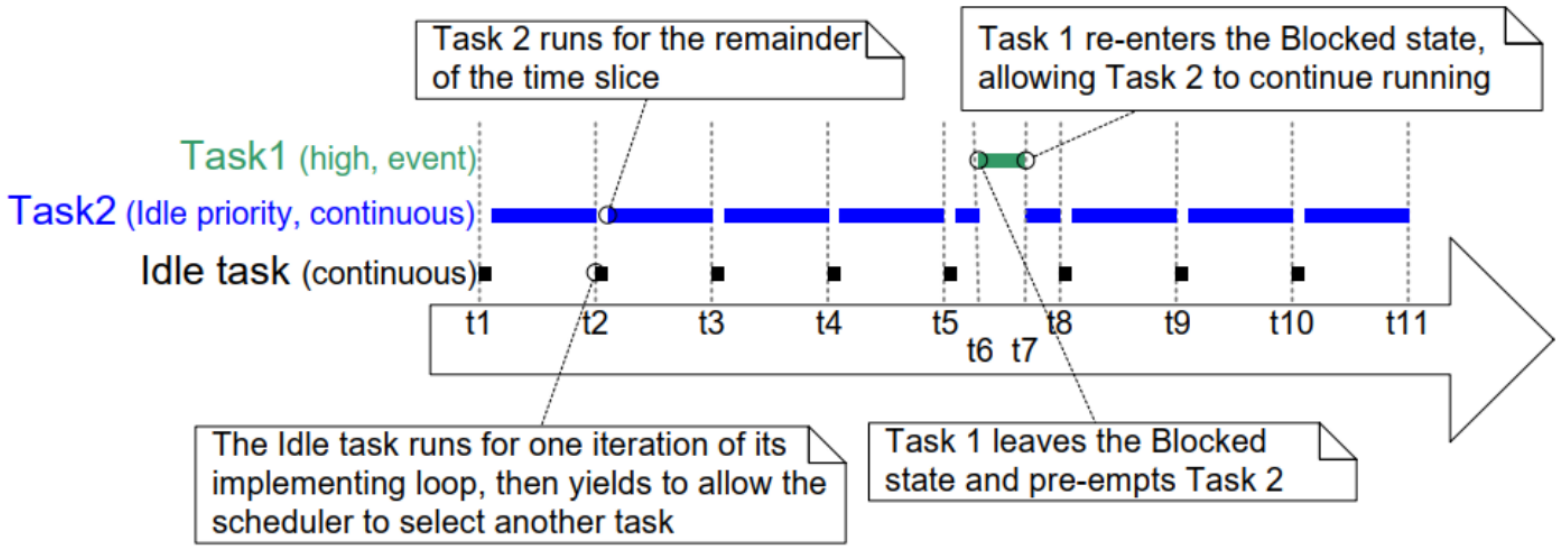


Preemption with continuous processing tasks



The Idle task sharing processing time with a task created by the application writer. Allocating that much processing time to the Idle task might not be desirable if the Idle priority tasks created by the application writer have work to do, but the Idle task does not. The configIDLE_SHOULD_YIELD compile time configuration constant can be used to change how the Idle task is scheduled.

Preemption with IDLE_SHOULD_YIELD



Selecting task priorities.

- **As a general rule, tasks that implement hard real-time functions are assigned priorities above those that implement soft real-time functions.** However, other characteristics, such as execution times and processor utilization, must also be taken into account to ensure the entire application will never miss a hard real-time deadline.
- Rate Monotonic Scheduling (RMS) is a common priority assignment technique which dictates that **a unique priority be assigned to each task in accordance with the tasks periodic execution rate**.

Scheduling algorithms

The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.

The algorithm can be changed using the configUSE_PREEMPTION and configUSE_TIME_SLICING configuration constants.

A third configuration constant, configUSE_TICKLESS_IDLE, also affects the scheduling algorithm, as its use can result in the tick interrupt being turned off completely for extended periods: it is an advanced option provided specifically for use in applications that must minimize their power consumption.

Scheduling algorithms

Scheduling algorithm	**Prioritized**	USE PREEMPTION	USE TIME SLICING
Preempt. with timeslicing	Yes	1	1
Preempt. without timeslicing	Yes	1	0
Co-operative	No	0	Any

Scheduling algorithms

Scheduling algorithms described as **Fixed Priority** do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority or that of other tasks.

Scheduling algorithms

Preemptive scheduling algorithms will immediately 'preempt' the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being preempted means being involuntarily moved out of the Running state and into the Ready state (without explicitly yielding or blocking) to allow a different task to enter the Running state. **Task preemption can occur at any time, not just in the RTOS tick interrupt**.

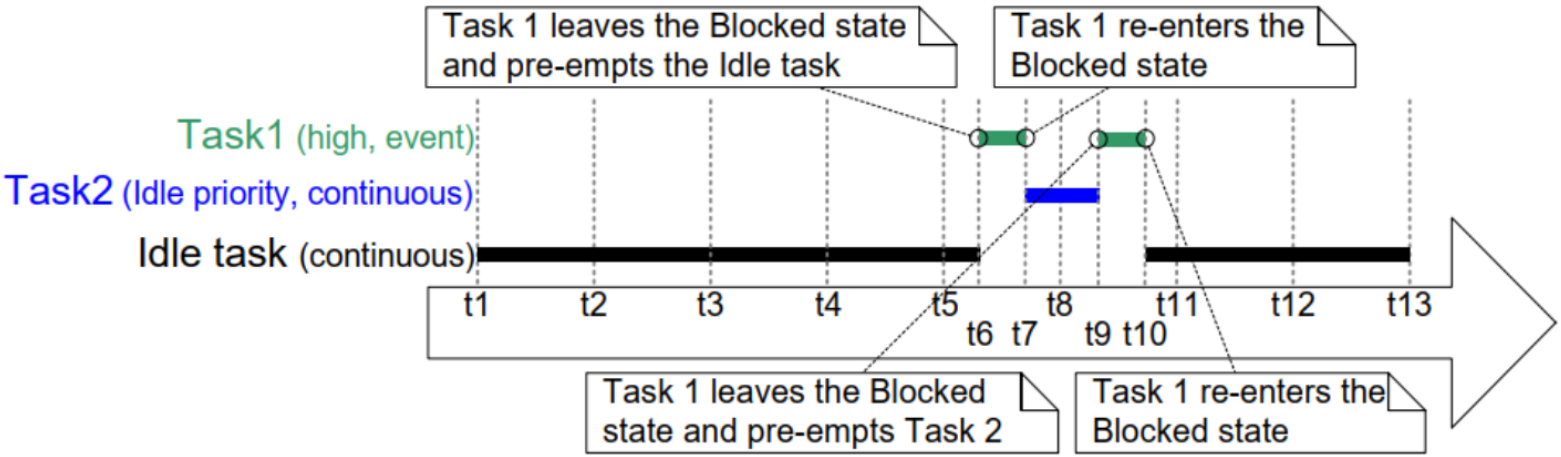
Scheduling algorithms

Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using Time Slicing select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Without time slicing

Prioritized Preemptive Scheduling without time slicing maintains the same task selection and preemption algorithms as described in the previous section, **but does not use time slicing to share processing time between tasks of equal priority**.
There are fewer task context switches when time slicing is not used than when time slicing is used. Therefore, turning time slicing off results in a reduction in the scheduler's processing overhead.
However, turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time.

Without time slicing

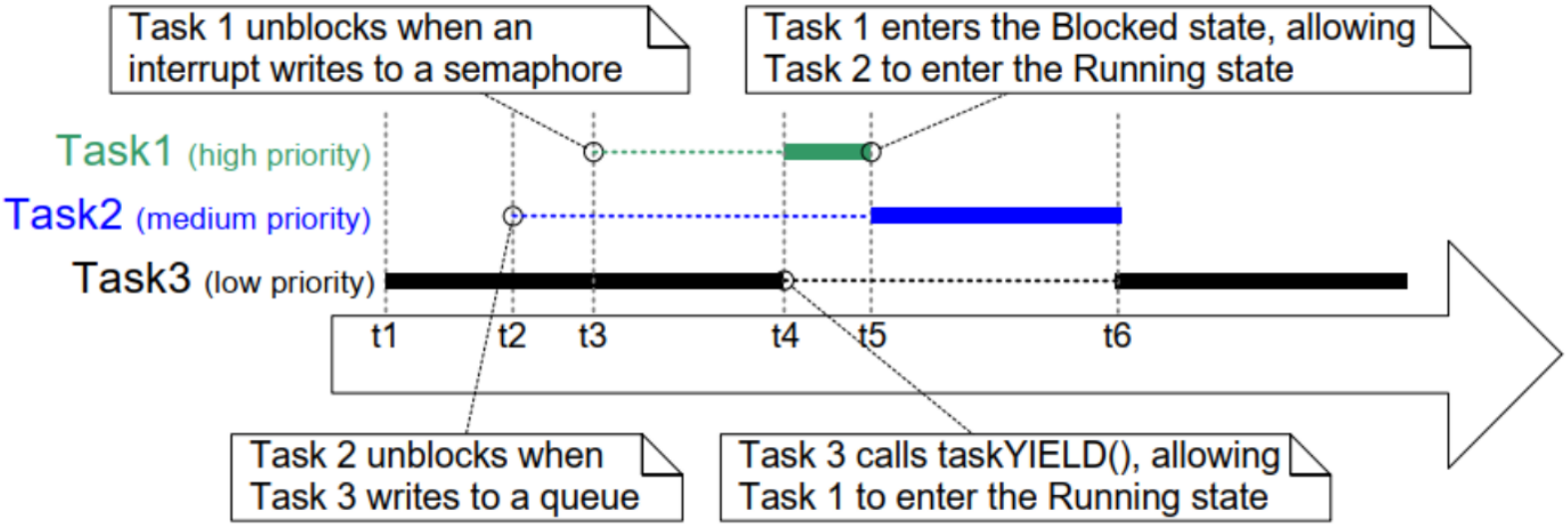


Cooperative scheduling

FreeRTOS can also optionally use co-operative scheduling.

- When a pure co-operative scheduler is used, a context switch will occur only when either the Running state task enters the Blocked state or the Running state task explicitly calls **taskYIELD()**.
- Tasks will never be pre-empted and tasks of equal priority will not automatically share processing time.
- Co-operative scheduling in this manner is simpler but can potentially result in a less responsive system.

Cooperative scheduling



Queue: introduction

Applications that use FreeRTOS are structured as a set of independent tasks—each task is effectively a mini program in its own right.
It is likely that these autonomous tasks will have to communicate with each other so that, collectively, they can provide useful system functionality. The 'queue' is the underlying primitive used by all FreeRTOS communication and synchronization mechanisms.

Queues

- A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.
- Normally, queues are used as First In First Out (FIFO) buffers where data is written to the end (tail) of the queue and removed from the front (head) of the queue. It is also possible to write to the front of a queue.
- **Writing data to a queue causes a byte-for-byte copy of the data to be stored in the queue itself. Reading data from a queue causes the copy of the data to be removed from the queue.**

Queues: access by tasks

Queues are objects in their own right that are not owned by or assigned to any particular task.

Queues: access by tasks

Queues are objects in their own right that are not owned by or assigned to any particular task.
Any number of tasks can write to the same queue and any number of tasks can read from the same queue.

Queues: access by tasks

Queues are objects in their own right that are not owned by or assigned to any particular task.
Any number of tasks can write to the same queue and any number of tasks can read from the same queue.
A queue having multiple writers is very common, whereas a queue having multiple readers is quite rare.

Queues: blocking on read

When a task attempts to read from a queue it can optionally specify a 'block' time. This is the time the task should be kept in the Blocked state to wait for data to be available from the queue should the queue already be empty.
A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue. The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Queues: blocking on read

Queues can have multiple readers so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data. **If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked**.

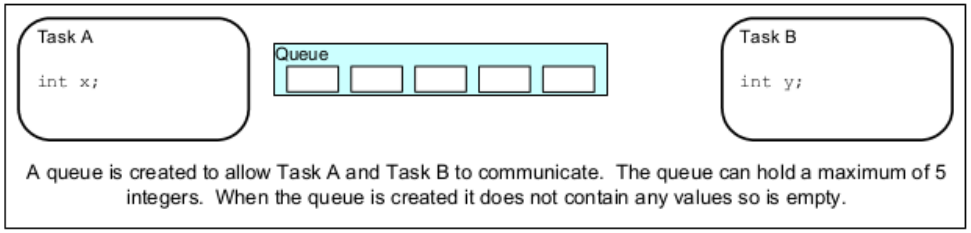
Queue: blocking on write

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.

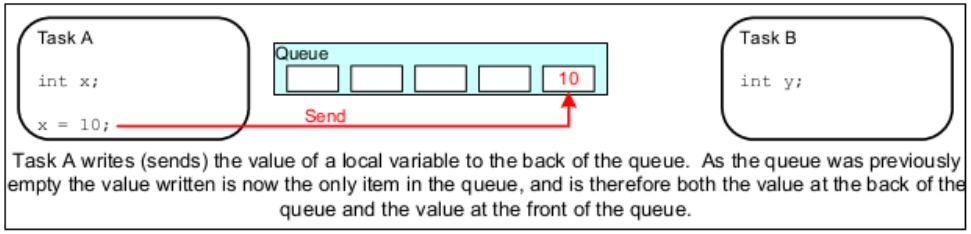
Queue: blocking on write

Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task that is waiting for space. **If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked.**

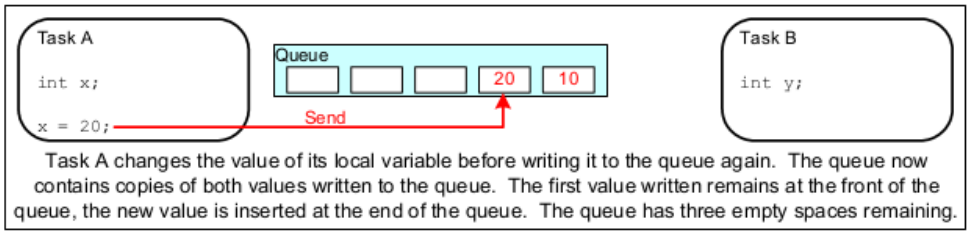
Queue



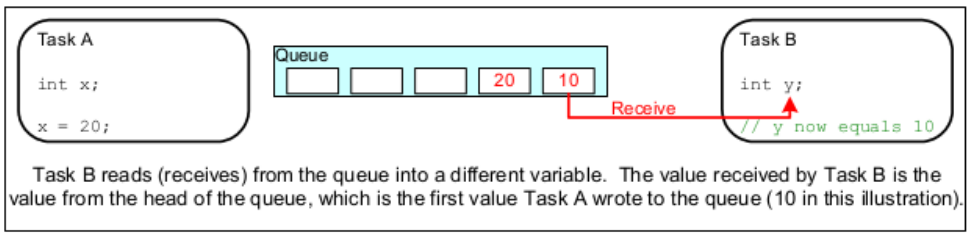
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



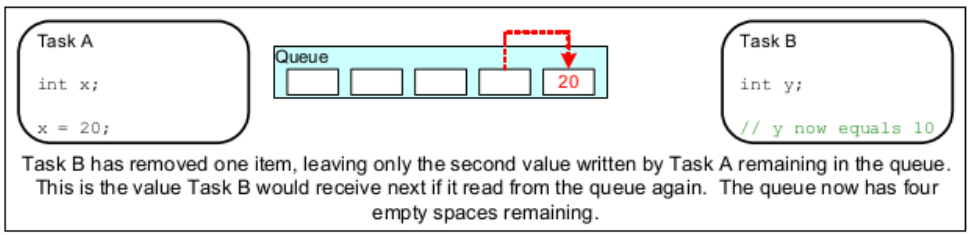
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Using a queue

- A queue must be explicitly created before it can be used.
- Queues are referenced using variables of type xQueueHandle. xQueueCreate() is used to create a queue and returns an **xQueueHandle** to reference the queue it creates.
- FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. The RAM is used to hold both the queue data structures and the items that are contained in the queue. xQueueCreate() will return NULL if there is insufficient heap RAM available for the queue to be created.

Using a queue: xQueueCreate

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize );
```

uxQueueLength The maximum number of items that the queue being created can hold at any one time.

uxItemSize The size in bytes of each data item that can be stored in the queue.

return If NULL is returned, then the queue cannot be created because there is insufficient heap memory available.

Using a queue: xQueueSendToBack, xQueueSendToFront

As might be expected, xQueueSendToBack() is used to send data to the back (tail) of a queue, and xQueueSendToFront() is used to send data to the front (head) of a queue.
xQueueSend() is equivalent to and exactly the same as xQueueSendToBack().

Interrupt routine

Never call xQueueSendToFront() or xQueueSendToBack() from an interrupt service routine. The interrupt-safe versions xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() should be used in their place.

Using a queue: xQueueSendToBack, xQueueSendToFront

```
portBASE_TYPE xQueueSendToFront(xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait);
```

```
portBASE_TYPE xQueueSendToBack(xQueueHandle xQueue,
                               const void * pvItemToQueue,
                               portTickType xTicksToWait);
```

xQueue The handle of the queue to which the data is being sent (written).

pvItemToQueue A pointer to the data to be copied into the queue.

Using a queue: xQueueSendToBack, xQueueSendToFront

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue. Value portMAX_DELAY will cause the task to wait indefinitely, provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

return pdPASS will be returned only if data was successfully sent to the queue. errQUEUE_FULL will be returned if data could not be written to the queue because the queue was already full.

Using a queue: xQueueReceive, xQueuePeek

- xQueueReceive() is used to receive (read) an item from a queue. The item that is received is removed from the queue.
- xQueuePeek() is used to receive an item from a queue **without the item being removed** from the queue.

Interrupt routine

Never call xQueueReceive() or xQueuePeek() from an interrupt service routine. Use interrupt-safe xQueueReceiveFromISR().

Using a queue: xQueueReceive, xQueuePeek

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
                            const void * pvBuffer,
                            portTickType xTicksToWait)
```

```
portBASE_TYPE xQueuePeek(xQueueHandle xQueue,
                         const void * pvBuffer,
                         portTickType xTicksToWait)
```

xQueue The handle of the queue from which the data is being received (read).

pvBuffer A pointer to the memory into which the received data will be copied.

Using a queue: xQueueReceive, xQueuePeek

xTickToWait The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue. If xTicksToWait is zero, then both xQueueReceive() and xQueuePeek() will return immediately if the queue is already empty. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely.

return pdPASS will be returned only if data was successfully read from the queue. errQUEUE_EMPTY will be returned if data cannot be read from the queue because the queue is already empty.

uxQueueMessagesWaiting()

uxQueueMessagesWaiting() is used to query the number of items that are currently in a queue.

Interrupt routine

Never call uxQueueMessagesWaiting() from an interrupt service routine. The interrupt-safe uxQueueMessagesWaitingFromISR() should be used in its place.

uxQueueMessagesWaiting()

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

xQueue The handle of the queue being queried.

return The number of items that the queue being queried is currently holding.

Example 10

This example demonstrates a queue being created, data being sent to the queue from multiple tasks, and data being received from the queue. The queue is created to hold data items of type long. The tasks that send to the queue do not specify a block time, whereas the task that receives from the queue does.
The priority of the tasks that send to the queue is lower than the priority of the task that receives from the queue. **This means that the queue should never contain more than one item because, as soon as data is sent to the queue the receiving task will unblock, pre-empt the sending task, and remove the data—leaving the queue empty once again.**

Example 10

```
static void vSenderTask( void *pvParameters )
{
    long lValueToSend;
    portBASE_TYPE xStatus;
    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value. The queue was created to hold values of type long,
    so cast the parameter to the required type. */
    lValueToSend = ( long ) pvParameters;
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS )
        {
        /* The send operation could not complete because the queue was full -
        this must be an error as the queue should never contain more than
        one item! */
        vPrintString( "Could not send to the queue.\n" );
        }
        /* Allow the other sender task to execute. taskYIELD() informs the
        scheduler that a switch to another task should occur now rather than
        keeping this task in the Running state until the end of the current time
        slice. */
        taskYIELD();
    }
}
```

Example 10

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the variable that will hold the values received from the queue. */
    long lReceivedValue;
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will
        immediately remove any data that is written to the queue. */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\n" );
        }
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
```

Example 10

```
        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* Data was not received from the queue even after waiting for 100ms.
            This must be an error as the sending tasks are free running and will be
            continuously writing to the queue. */
            vPrintString( "Could not receive from the queue.\n" );
        }
    }
}
```

Example 10

```
/* Declare a variable of type xQueueHandle. This is used to store the handle
to the queue that is accessed by all three tasks. */
xQueueHandle xQueue;
int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type long. */
    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue. The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue. Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 240, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, ( void * ) 200, 1, NULL );
        /* Create the task that will read from the queue. The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 2, NULL );
        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
```

Example 10

```
    else
    {
        /* The queue could not be created. */
    }
    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    */
    for( ;; );
}
```

The tasks that send to the queue call taskYIELD() on each iteration of their infinite loop. taskYIELD() informs the scheduler that a switch to another task should occur now, rather than keeping the executing task in the Running state until the end of the current time slice. A task that calls taskYIELD() is in effect volunteering to be removed from the Running state.

Example 10: timing diagram

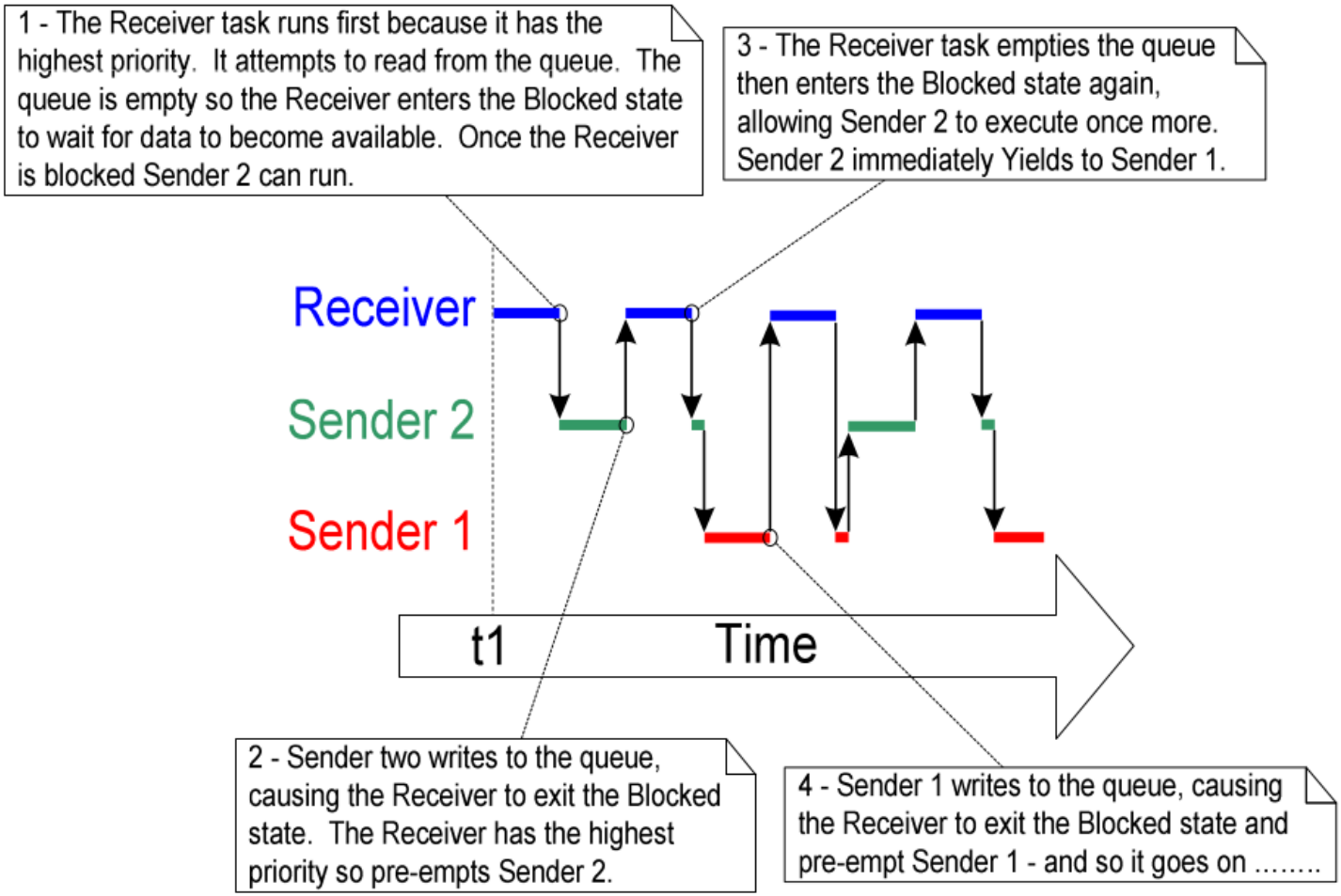


Figure: Example 10

Transfer Compound Types

It is common for a task to receive data from multiple sources on a single queue. Often, the receiver of the data needs to know where the data came from, to allow it to determine how the data should be processed. A simple way to achieve this is to use the queue to transfer structures where both the value of the data and the source of the data are contained in the structure fields.

Compound types

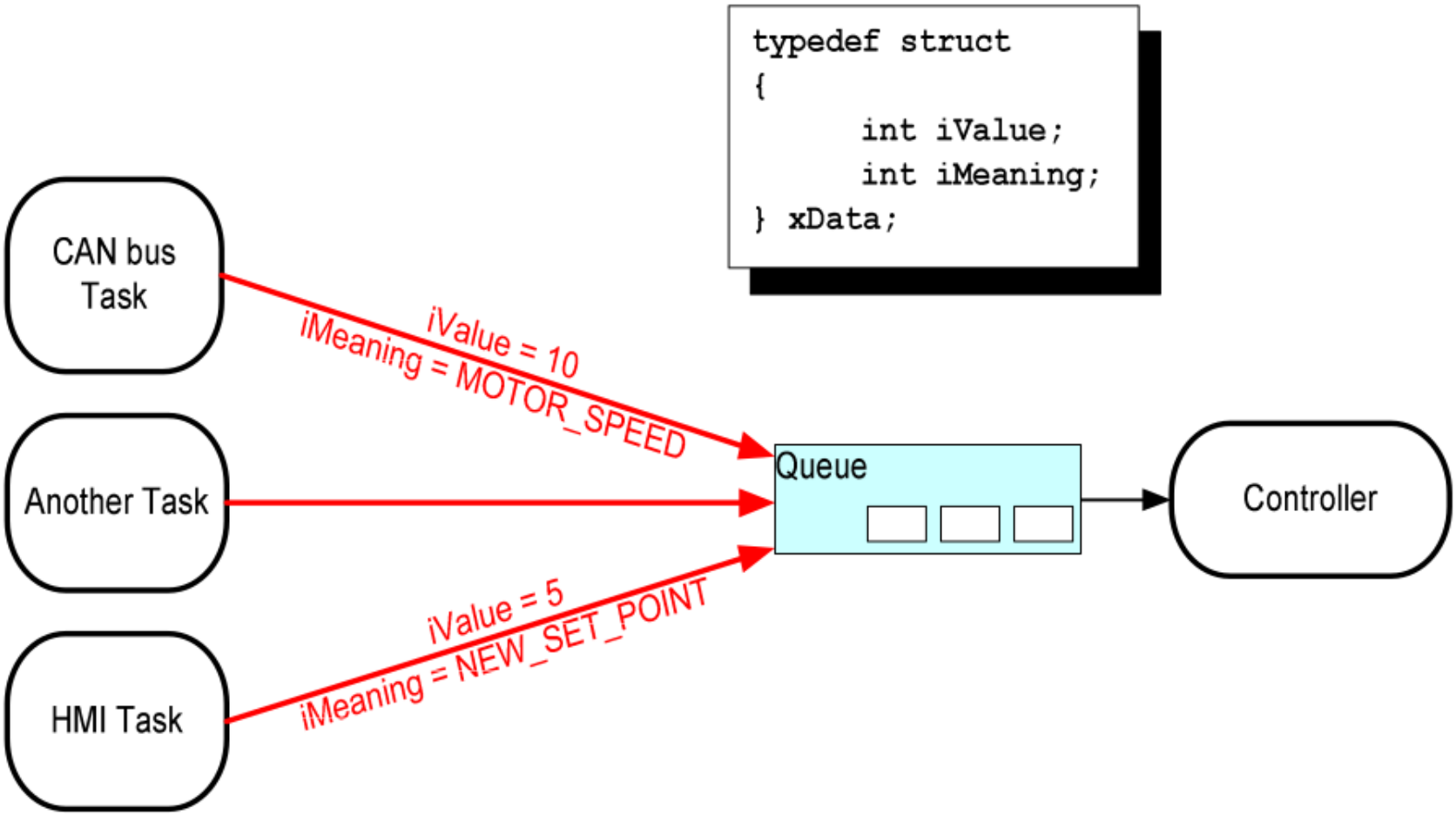


Figure: Compound type example.

Example 11

Example 11 is similar to Example 10, but the task priorities are reversed so the receiving task has a lower priority than the sending tasks. Also the queue is used to pass structures, rather than simple long integers, between the tasks.

Example 11

```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;
/* Declare two variables of type xData that will be passed on the queue. */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Used by Sender1. */
    { 200, mainSENDER_2 } /* Used by Sender2. */
};
```

In Example 11, the sending tasks have the higher priority, so the queue will normally be full. This occurs because, as soon as the receiving task removes an item from the queue, it is pre-empted by one of the sending tasks which then immediately re-fills the queue. The sending task then re-enters the Blocked state to wait for space to become available on the queue again.

Example 11

```
static void vSenderTask( void *pvParameters )
{
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send to the queue.
        The second parameter is the address of the structure being sent. The
        address is passed in as the task parameter so pvParameters is used
        directly.
        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        if the queue is already full. A block time is specified because the
        sending tasks have a higher priority than the receiving task so the queue
        is expected to become full. The receiving task will remove items from
        the queue when both sending tasks are in the Blocked state. */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );
        if( xStatus != pdPASS )
        {
        /* The send operation could not complete, even after waiting for 100ms.
        This must be an error as the receiving task should make space in the
        queue as soon as both sending tasks are in the Blocked state. */
            vPrintString( "Could not send to the queue.\n" );
        }
        /* Allow the other sender task to execute. */
        taskYIELD();
    }
}
```

Example 11

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the structure that will hold the values received from the queue. */
    xData xReceivedStructure;
    portBASE_TYPE xStatus;
    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\n" );
        }

        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value and the source of the value. */
            if( xReceivedStructure.ucSource == mainSENDER_1 )
            {
            vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
            vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
```

Example 11

```
            else
            {
            vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
        /* Nothing was received from the queue. This must be an error
        as this task should only run when the queue is full. */
            vPrintString( "Could not receive from the queue.\n" );
        }
    }
}
```

Example 11

```
int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type xData. */
    xQueue = xQueueCreate( 3, sizeof( xData ) );
    if( xQueue != NULL )
    {
        /* Create two instances of the task that will write to the queue. The
        parameter is used to pass the structure that the task will write to the
        queue, so one task will continuously send xStructsToSend[ 0 ] to the queue
        while the other task will continuously send xStructsToSend[ 1 ]. Both tasks
        are created at priority 2 which is above the priority of the receiver. */
        xTaskCreate( vSenderTask, "Sender1", 240, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, &( xStructsToSend[ 1 ] ), 2, NULL );
        /* Create the task that will read from the queue. The task is created with
        priority 1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 1, NULL );
        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }
    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created. */
    for( ;; );
}
```

Example 11: timing diagram

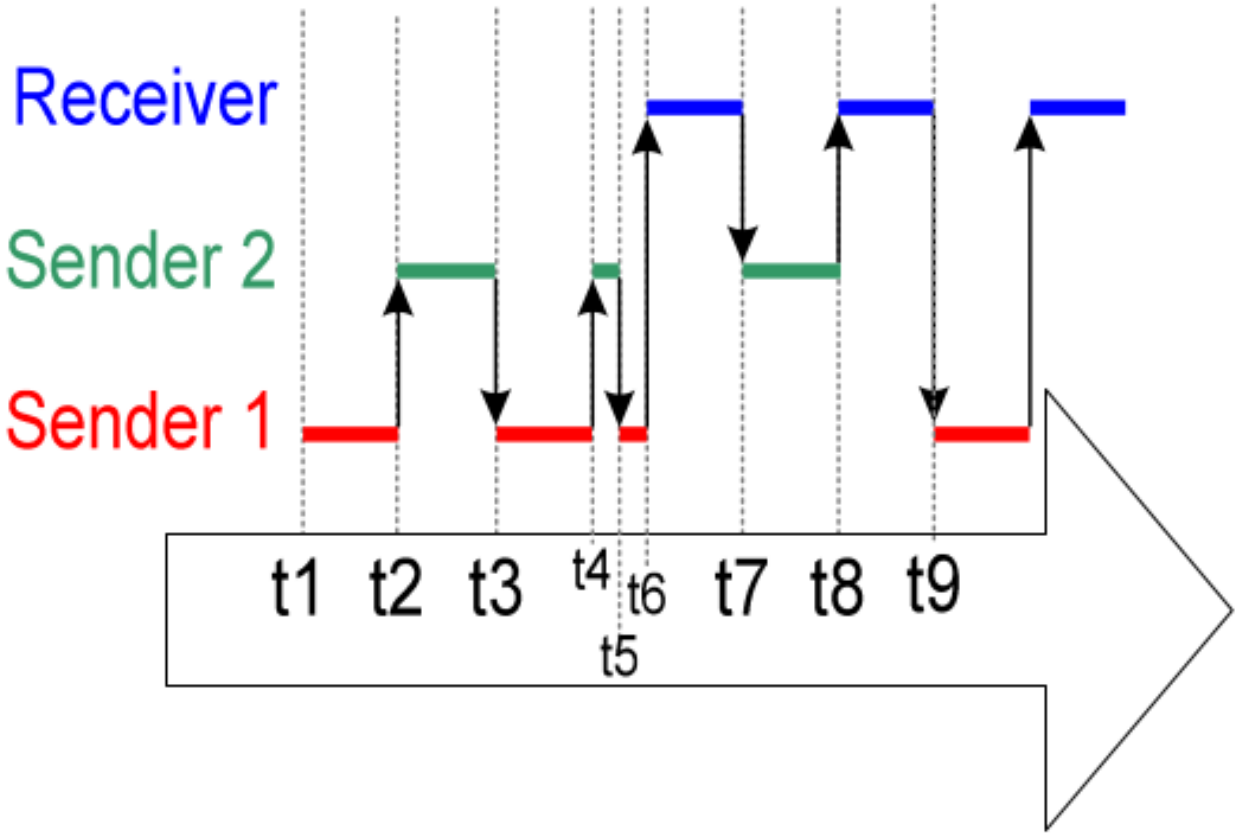


Figure: Example 11

Working with large data

If the size of the data being stored in the queue is large, then it is preferable to use the queue to transfer pointers to the data, rather than copy the data itself into and out of the queue byte by byte. Transferring pointers is more efficient in both processing time and the amount of RAM required to create the queue. However, when queuing pointers, extreme care must be taken to ensure that:

- The owner of the RAM being pointed to is clearly defined. **When sharing memory between tasks via a pointer, it is essential to ensure that both tasks do not modify the memory contents simultaneously.**

Working with large data

- The RAM being pointed to remains valid. If the memory being pointed to was allocated dynamically, **then exactly one task should be responsible for freeing the memory**. No task should attempt to access the memory after it has been freed.

Pointer to stack data

A pointer should never be used to access data that has been allocated on a task stack. The data will not be valid after the stack frame has changed.

Alternative usage of queue objects

- Queue set with xQueueCreateSet, xQueueAddToSet and xQueueSelectFromSet.
- Mailbox with xQueuePeek and xQueueOverwrite