ARM® RTX™
Real-Time Operating System

A Cortex-M Optimized RTOS that Simplifies Embedded Programming

Summer/Winter 2013

Bob Boys
ARM
San Jose, California
bob.boys@arm.com

Agenda

Agenda:

- CMSIS
- Super-loop vs. RTOS
- Round Robin RTOS
- Create a Project
- Debugging
- Cooperative Multitasking
- RTOS Objects
- Semaphore Example
- Interrupts
- Preemptive Task Switching

- It is all about productivity !

CMSIS: Cortex Microcontroller Software Interface Standard.

- CMSIS-Core Version 3.0
  - Standard header files for various Cortex-M processors.
  - Simplify startup routines. Startup.s and system.c
  - System View Description (SVD) XML files. (peripherals)
- CMSIS-DSP
  - Digital Signal processing libraries
  - Free for Cortex-M0, M3 and M4
- CMSIS-RTOS
  - A standard API for RTOSs
  - Includes RTX – a BSD license … this means it if free…
- CMSIS-DAP
  - A standard to connect "on-board debug adapters"
  - Uses USB and a small processor on the PC board.

OTHER RTOS

- Linux
- Android
- FreeRTOS
- WinCE
- Micrium, ExpressLogic, Quadros, QNX and so on.
- RTX

LICENSES

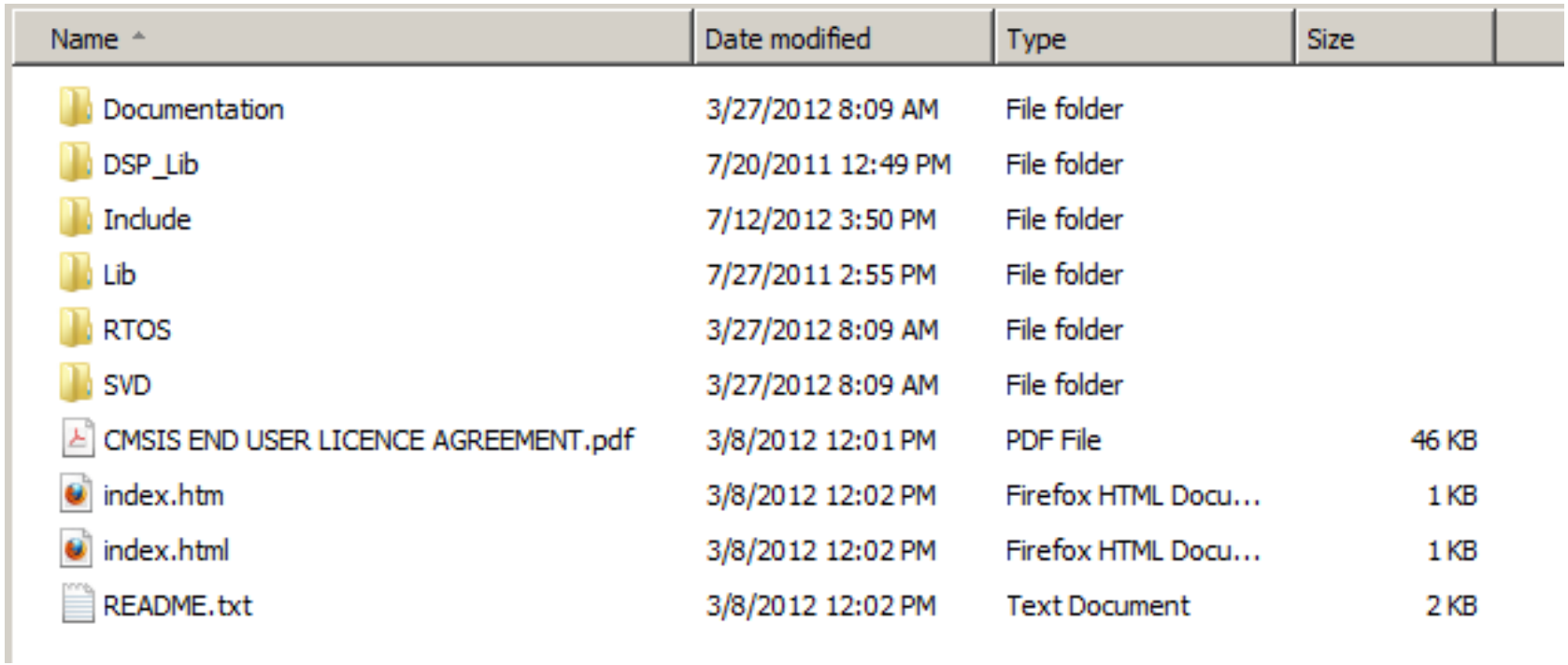
I am not a lawyer and this is not legal advice of any kind.

- Some things you should look into for commercial products:
- **GPL:** a complicated license. Must return your changes and improvements to the community. Must provide source code. **Lesser GPL:** permits use of a library in proprietary programs.
- **BSD:** Berkeley Software Distribution: essentially free.
- **MIT:** http://opensource.org/licenses/mit-license.php
- **Apache:** somewhat like BSD – For details see: www.apache.org/licenses/LICENSE-2.0.html
- **Public Domain**: no owner. Do with it as you like.
- Many/most companies have or hire a lawyer to advise them.

WHERE DID RTX COME FROM ?

- Developed by Reinhard Keil and his team in Germany.
- RTX and RTXTiny for 8051: made in Switzerland.
- RTX166 for Siemens C166 processors. Munich, Germany.
- RTX for ARM7 and ARM9.
  in Munich and Slovenia.
- RTX for Cortex-M3. Slovenia.
- Originally free with Keil ARM tools.
  Paid for source code.
- Now with BSD license.

FILES: WWW.KEIL.COM: MDK



Name	Date modified	Type	Size
Documentation	3/27/2012 8:09 AM	File folder	
DSP_Lib	7/20/2011 12:49 PM	File folder	
Include	7/12/2012 3:50 PM	File folder	
Lib	7/27/2011 2:55 PM	File folder	
RTOS	3/27/2012 8:09 AM	File folder	
SVD	3/27/2012 8:09 AM	File folder	
CMSIS END USER LICENCE AGREEMENT.pdf	3/8/2012 12:01 PM	PDF File	46 KB
index.htm	3/8/2012 12:02 PM	Firefox HTML Docu...	1 KB
index.html	3/8/2012 12:02 PM	Firefox HTML Docu...	1 KB
README.txt	3/8/2012 12:02 PM	Text Document	2 KB

Forums.arm.com See the cmsis category.

www.keil.com/demo/eval/rtx.htm

CMSIS is created and maintained by ARM

HOW TO TRY OUT CMSIS:

- www.keil.com/arm
- download free Keil MDK toolset
- Comes with simulator.
- Or connect to a target with Debug adapter: Keil ULINK, ST-Link V 2 or Segger J-Link (black case V 6 or later)
- Need Serial Wire Viewer support for RTOS:
- Many examples are provided.
- See my labs…. www.keil.com
- Ports for RTX: Keil MDK, GCC and IAR
- www.keil.com/demo/eval/rtx.htm

Important Things in Cortex-M for RTOS

- Set/Unset hardware breakpoints on-the-fly.
- Dedicated SysTick timer.
- Serial Wire Viewer (SWV): see events real-time.
- ITM™ printf
- DAP read/write capability: Watch and Memory
- Kernel Awareness windows…
- Watchpoints: dedicated breakpoints.
- ETM™ trace: see all program counters.
- LDREX and STREX instructions.
- ….plus a few other things to make RTOS life easier…

A RTOS's Advantages Over a Superloop

Developers focus on writing their application

- Rather than the scheduling and timing of it
- Device drivers, I/O and interrupt interfaces included
- Easy to get working…

Enables logical partitioning of application

- Application code is more portable

Even simple applications need boot code, interrupt management and device drivers; Included within RTOS

- Buying an RTOS for this code alone is cheaper than writing it from scratch:
- **BECAUSE ITS FREE !!!!!!**

Keil RTX RTOS Advantages

Royalty Free
Small footprint
Fast interrupts
Optimized for Cortex M processors.
Kernel aware debugging in Keil µVision… (secret weapon)
CMSIS-RTOS compliant.

Task Specifications	
Priority Levels	**254**
No. of Tasks Defined	**Unlimited**
No. of Tasks Active	**250**
Context Switch	**< 200 Cycles**
Interrupt Latency	**Not Applicable for Cortex M**

Memory Requirements	Bytes
CODE Space (depending on used functionality)	**<4K**
RAM Space (each active task requires an own stack space)	**< 500**

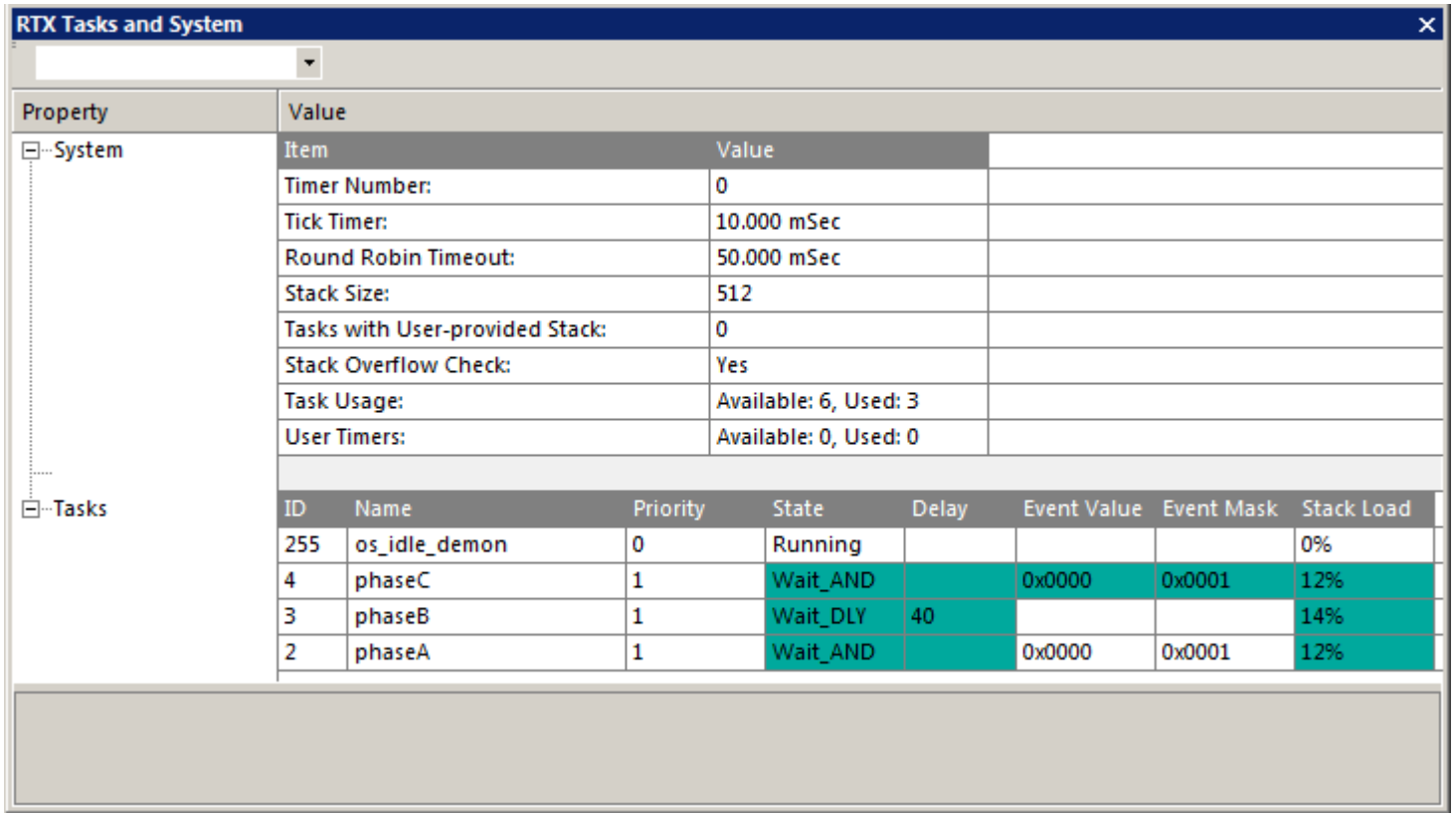
RTX Performance

Task Specifications	Cycles
Interupt Latency	n/a
Initialize system, start task	1,147
Create task (no task switch)	403
Create task (with task switch)	461
Delete Task	218
Task Switch	192
Set event	89
Send semaphore	72

RTX MODES

- Round Robin:
  - Each task given a time slice.
  - When slice is up – switches to next task.
  - If task not done – will complete it later.
- Pre-emptive:
  - A task runs until interrupted by a higher priority task. This is a "context switch"
  - First task will run later…
- Co-operative:
  - A task runs until gets a "blocking OS call" or uses a os_task_pass() call.

RTX TASKS AND SYSTEM VIEWER

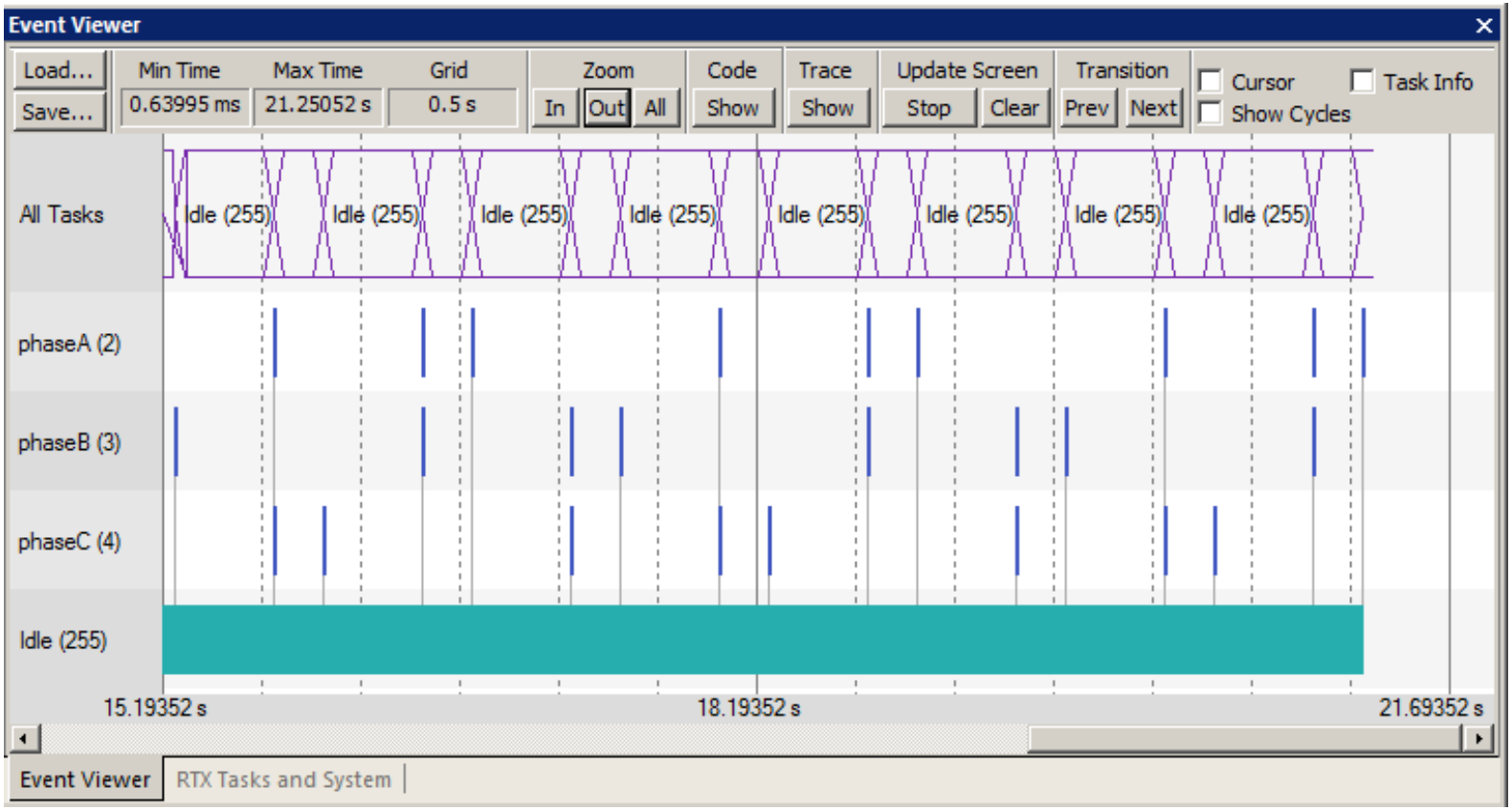


Item	Value
Timer Number:	0
Tick Timer:	10.000 mSec
Round Robin Timeout:	50.000 mSec
Stack Size:	512
Tasks with User-provided Stack:	0
Stack Overflow Check:	Yes
Task Usage:	Available: 6, Used: 3
User Timers:	Available: 0, Used: 0

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Running				0%
4	phaseC	1	Wait_AND		0x0000	0x0001	12%
3	phaseB	1	Wait_DLY	40			14%
2	phaseA	1	Wait_AND		0x0000	0x0001	12%

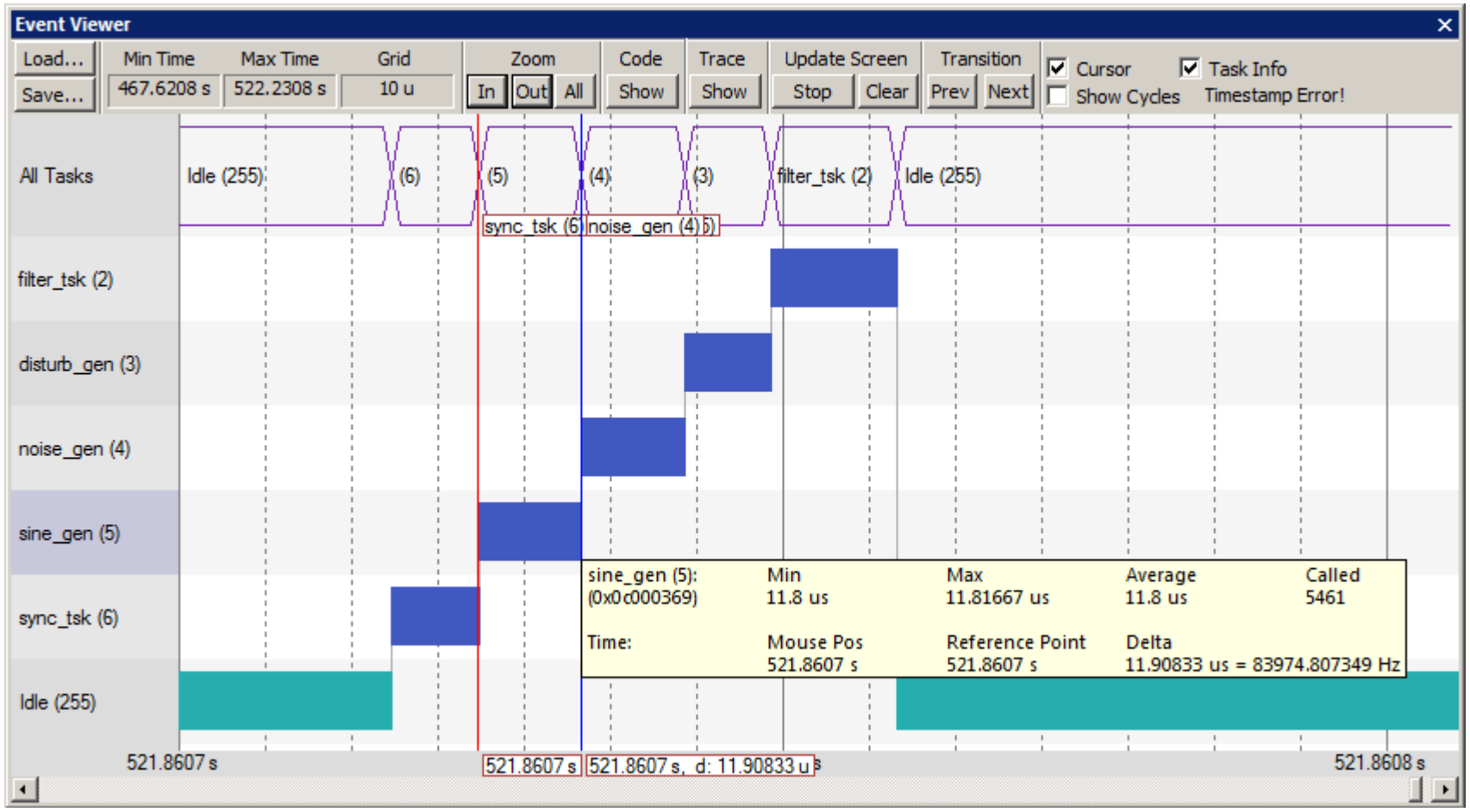
- Uses DAP R/W if using a real chip –
- Shows what is running and waiting etc.
- Updates in real-time.

RTX EVENT VIEWER



- With the Simulator or if chip uses SWV.
- Updates in real-time.
- Shows when tasks operating.

RTX EVENT VIEWER:



- Can zoom in to look at times.
- Very useful for debugging and configuring RX...

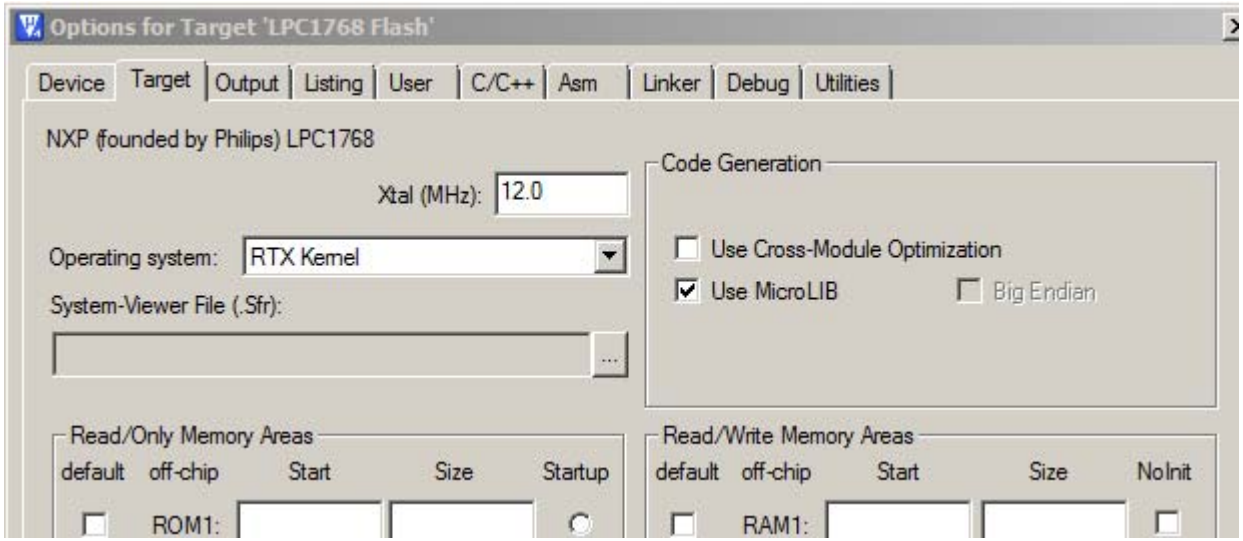
Most Popular Functions:

- All functions are in the RTX Help.
- Here are some of the most common ones:

- **NOTE:**
- Cortex-M use SysTick Timer. Is Interrupt 15.
- RTX uses SVCall instruction to access privileged resources. Is Interrupt 11.
- RTX will use special features of Cortex-M for most efficient operation.

First – some configuration items:

- #include <RTL.h>
- Select RTX Kernel in Options for target as shown below:



- OSTID t_phaseA; assigns task ID of Task A.
  - One for each task

Now, set up and start RTX:

- **Create and start task phaseA: (one for each task)**
  - t_phaseA = os_tsk_create (phaseA, 0);
    - *(task name, priority 0 to 254)*

- **Initialize RTX and start init**
  - os_sys_init(init);
    - *(the task to call)*

- **Self delete a task: useful when init task done:**
  - os_tsk_delete_self ();

Controlling Functions for Tasks:

- **Delays a task a set of clock ticks (of SysTick).**
  - os_dly_wait (8);
    - *(number of SysTick ticks)* (10 msec)
    - Task is put in WAIT_DLY state.
    - When Delay done, is put in READY state.

Events:

- **Send event signal to clock task:**
  - os_evt_set (0x0001, task);
    - *(bit pattern event flag number, OS_TID task name)*

- **Wait for an event flag 0x0001**
  - os_evt_wait_and (0x0001, 0xffff);
  - os_evt_wait_or (0x0001, 0xffff);
    - *(event flag number, timeout value)*
    - ***and**: waits for a number of event flags.*
    - ***or**: waits for one event flag.*

Mutex

- **Declare a mutex:**
  - OS_MUT(OS_ID Mutex name);
- **Initialize a Mutex:**
  - os_mut_init(OS_ID Mutex name);

- **Acquire a Mutex:**
- os_mut_wait(OS_ID Mutex name, timeout);
- **Release a Mutex:**
  - os_mut_release(OS_ID Mutex name, timeout);

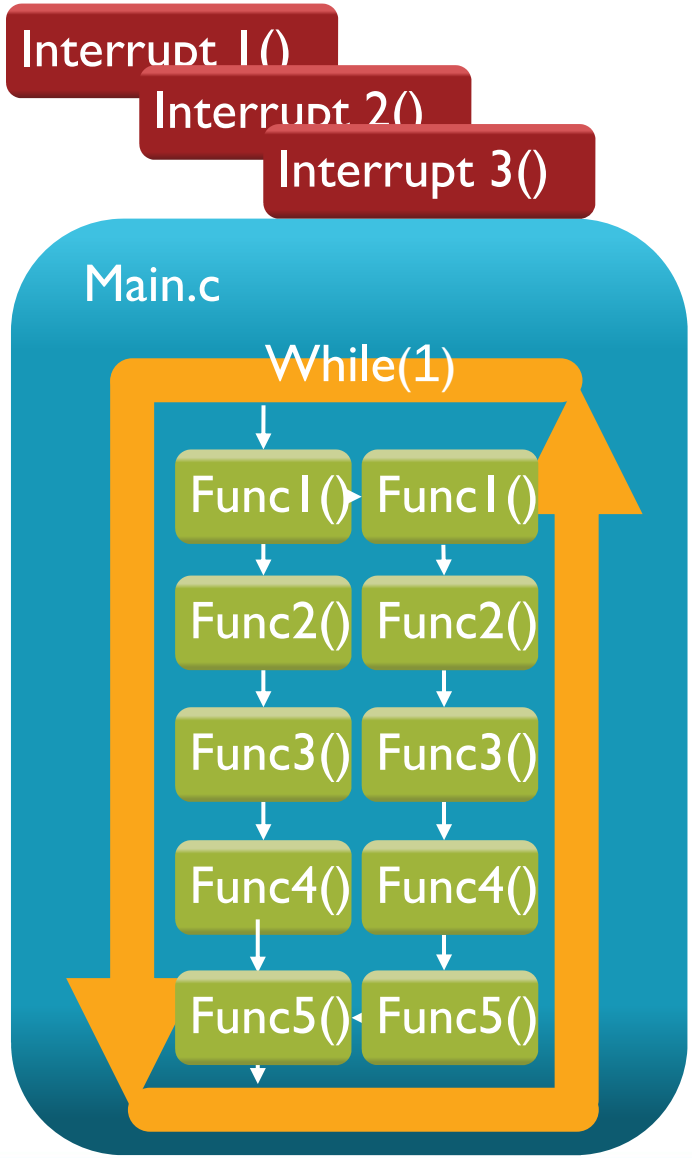
Super-Loop: Typical Programming Method

Application implemented as an endless loop with function calls

- Implies a fixed order for function execution
- Time-critical program areas use Interrupt Service Routines (ISR)s
- Communication via global variables; not data communication protocols

Perfect for small embedded systems

- Often used for 8/16-bit MCUs



Super-Loop Scaling Pitfalls - ISRs

Time-critical operations must be processed within ISRs

- ISR functions get complex and require long execution times
- ISR nesting may create unpredictable execution time and stack loads

Super-Loop Scaling Pitfalls - Timers

Super-Loop can be synchronized with the System timer, but

- If system requires several different cycle times, it is hard to implement
- Typically only 4 hardware timers – so only 4 events

Creates software overhead – application becomes hard to understand

Super-Loop Scaling Pitfalls – Data

Data exchange Super-Loop/ISR via global shared variables

- Application programmer must ensure data consistency

```
Interrupt_high()  {
      Global_var = 100;
}
```

```
Interrupt_low()  {
      Global_var = 50;
}
```

```
Void modify_var (void)  {
      Global_var = 20;
}
```

```
Interrupt ()  {
      Sendchar ("A");
}
```

```
Void printer (void)  {
      Printf("bbbbbbbbbbb");
}
```

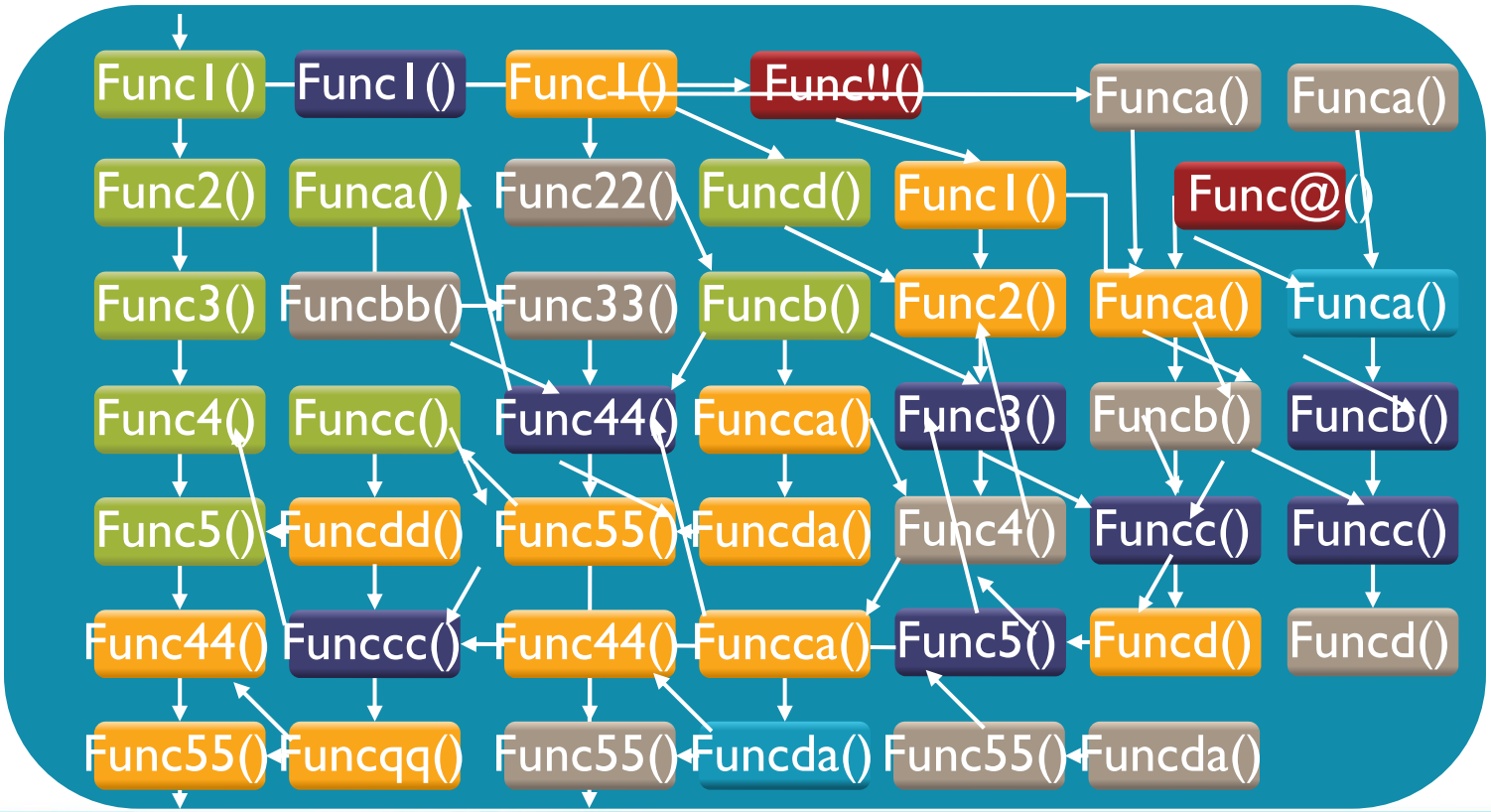
Serial Out

bbbbbbbAbbbbbb

Super-Loop Scaling Pitfalls – Scalability

Applications become complex and hard to extend

- A simple change can have unpredictable side effects
- Time consuming to analyze



Interrupt 1

Interrupt 2()

Interrupt 3

Using a RTOS

Giving Even More Control to Your Project

Using an RTOS

Programs are made up of multiple files and functions

Main.c

HighPri()
Init_periph()
Low_pwr_check()
Wait()
Master_switch()
Check_flags()
Check_clock()
Set_globals()
Interupt_ISRS()

Debug info()

IO.c

Check_ext_I()
Foo()
Bar()
Test()
Do_it()
Check_serial()
Check_input()
Check_ports()
Open_port()

Math.c

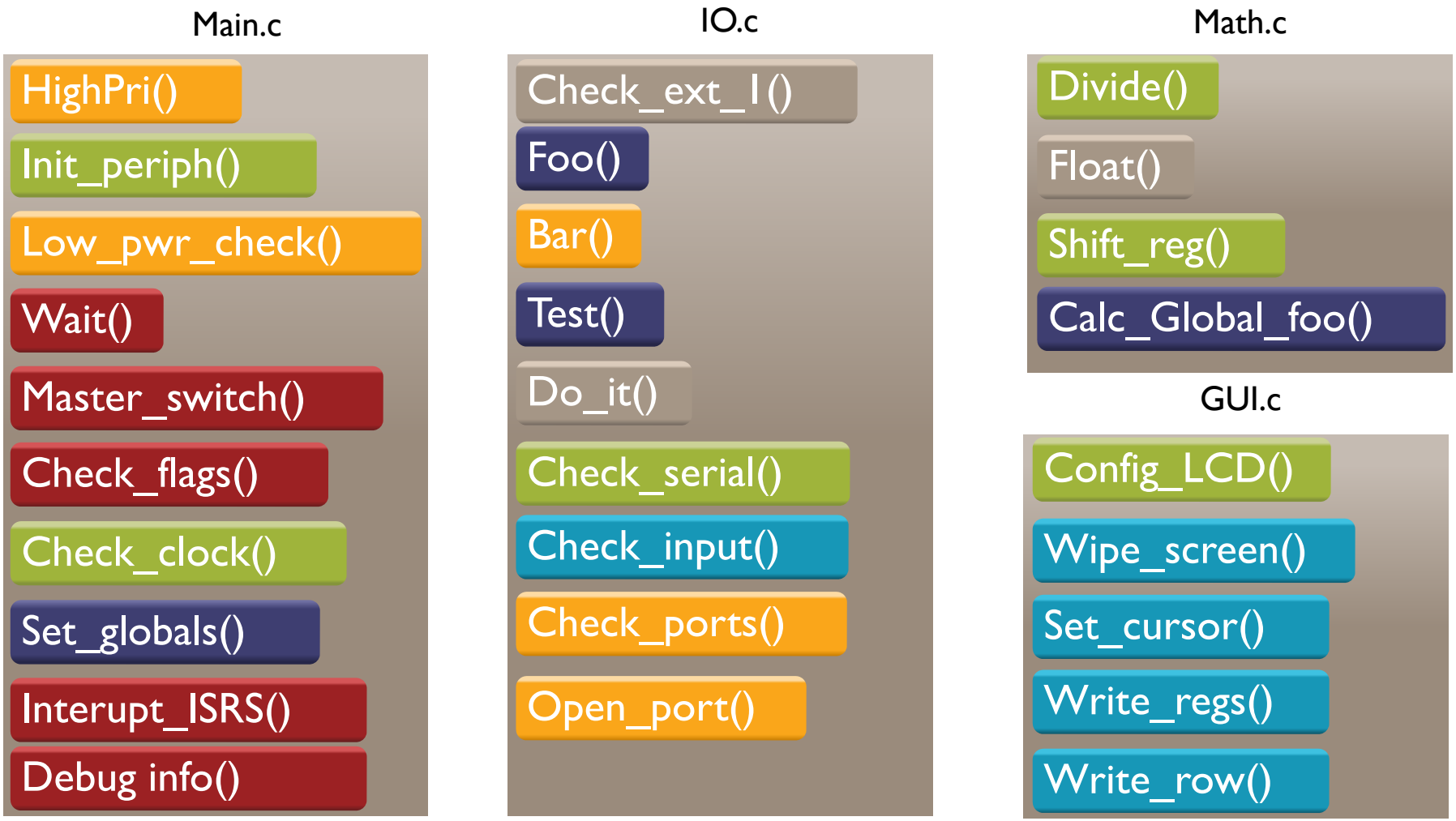
Divide()
Float()
Shift_reg()
Calc_Global_foo()

GUI.c

Config_LCD()
Wipe_screen()
Set_cursor()
Write_regs()
Write_row()

Using an RTOS

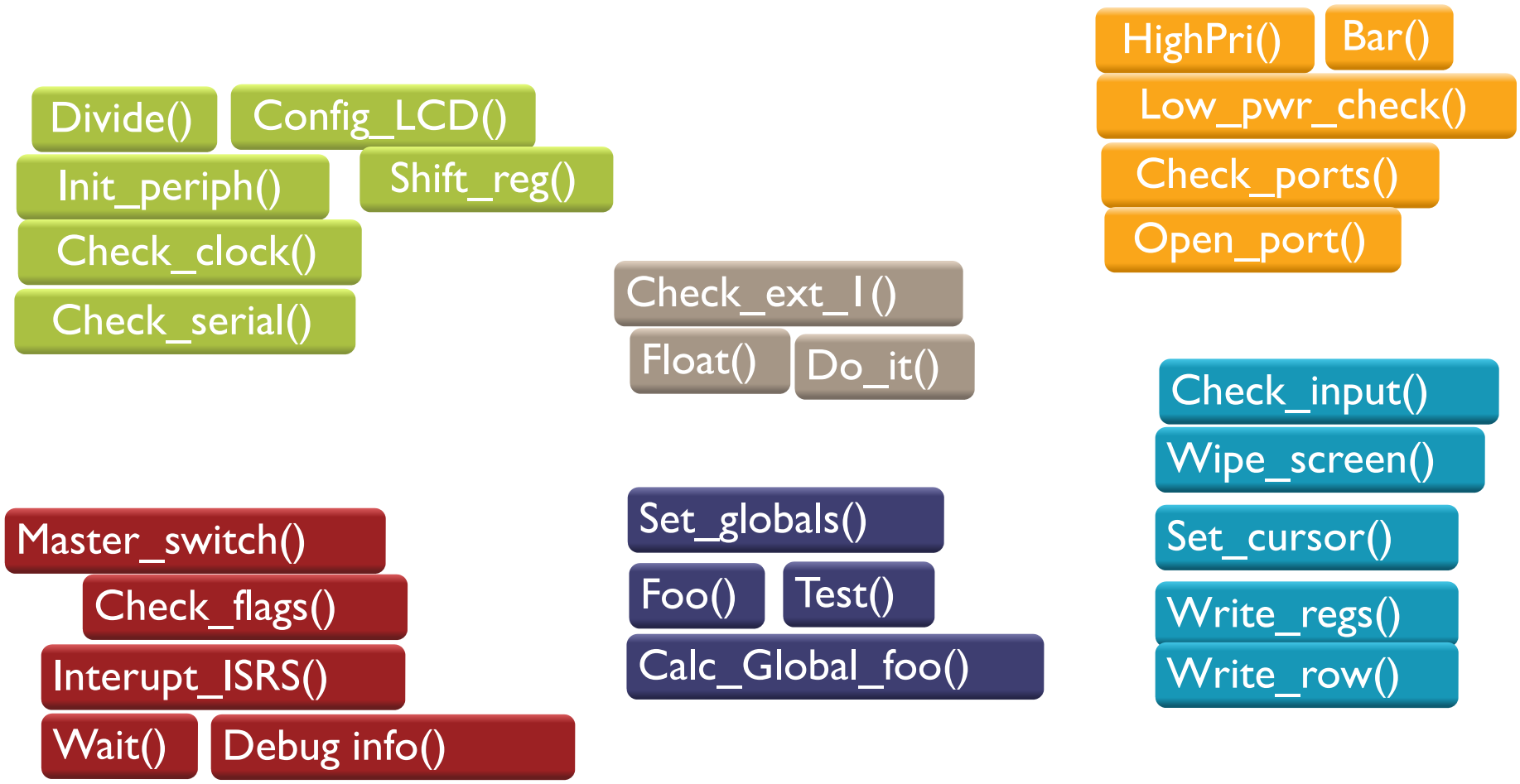
Functions can be grouped together logically



RTOS -Tasks

Logical Groups can be managed by an RTOS task

- Tasks can share functions



RTOS -Tasks

Logical Groups can be managed by an RTOS task

- Tasks can share functions

HighPri()
Bar()
Low_pwr_check()
Check_ports()
Open_port()

Check_input()
Wipe_screen()
Set_cursor()
Write_regs()
Write_row()

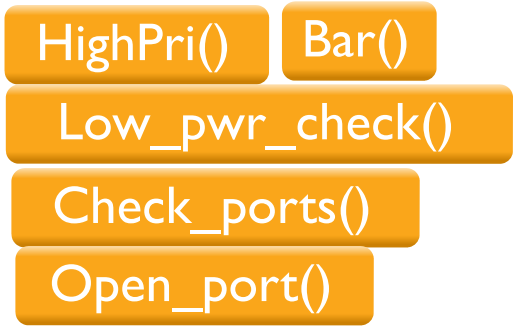
Master_switch()
Check_flags()
Interupt_ISRS()
Wait()
Debug info()

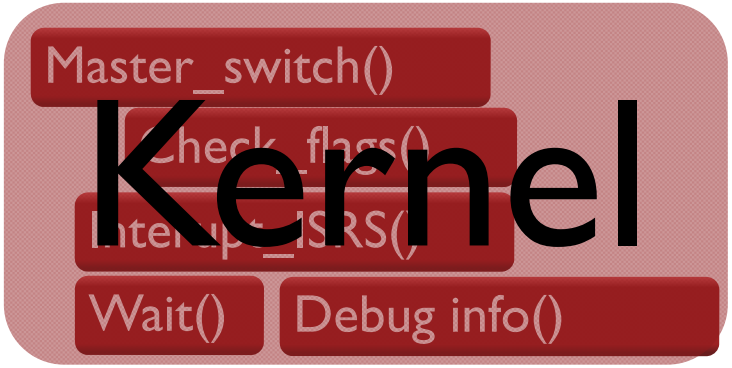
Set_globals()
Foo()
Test()
Calc_Global_foo()

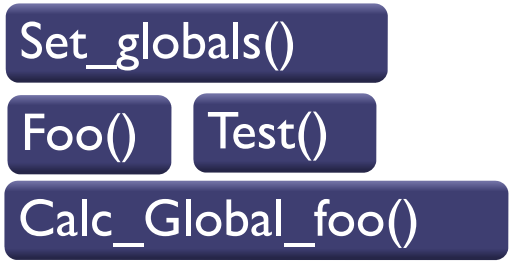
RTOS -Tasks

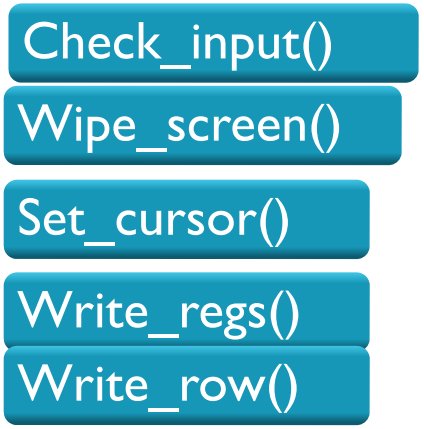
Logical Groups can be managed by an RTOS task

- Tasks can share functions





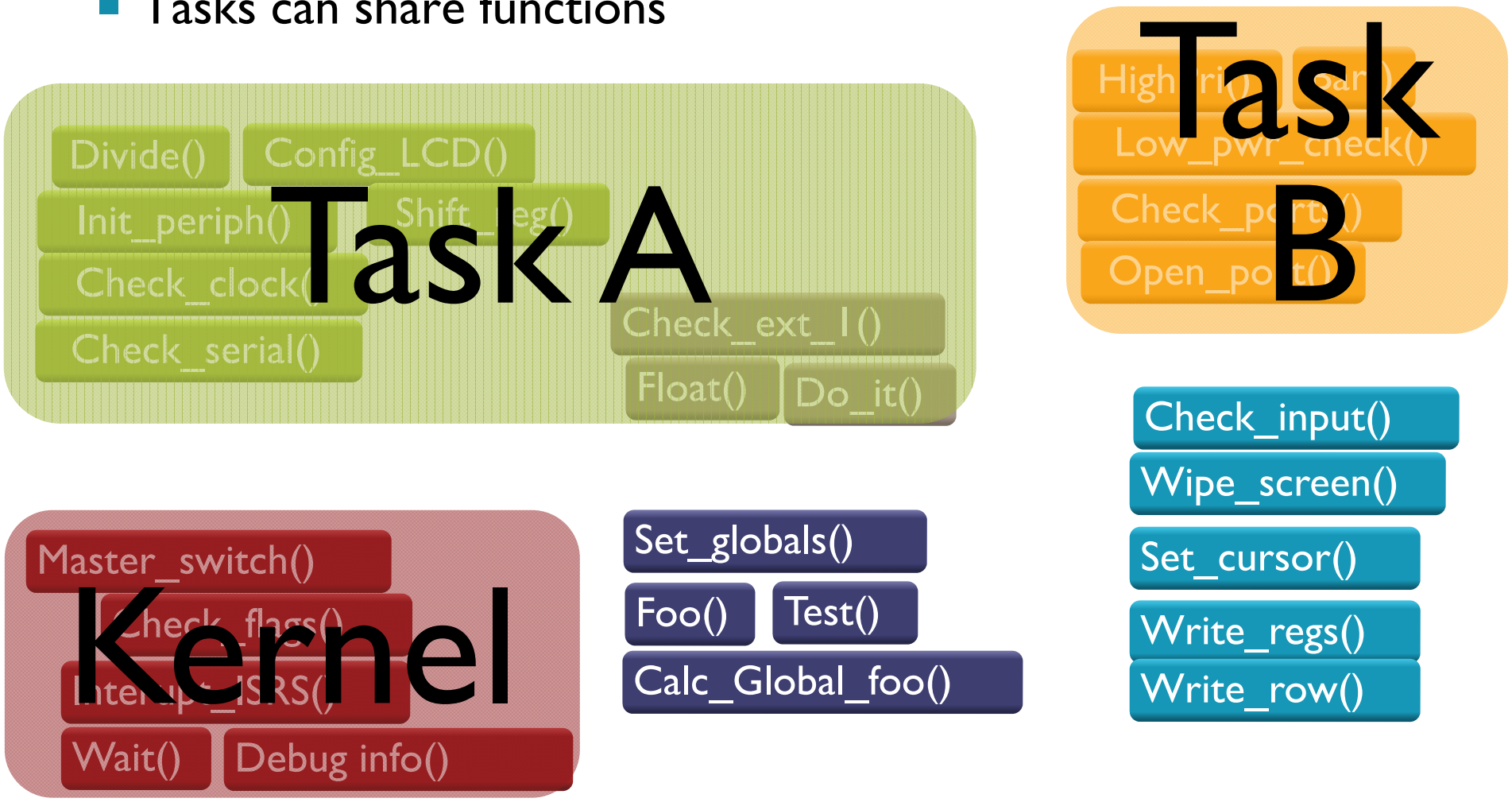




RTOS -Tasks

Logical Groups can be managed by an RTOS task

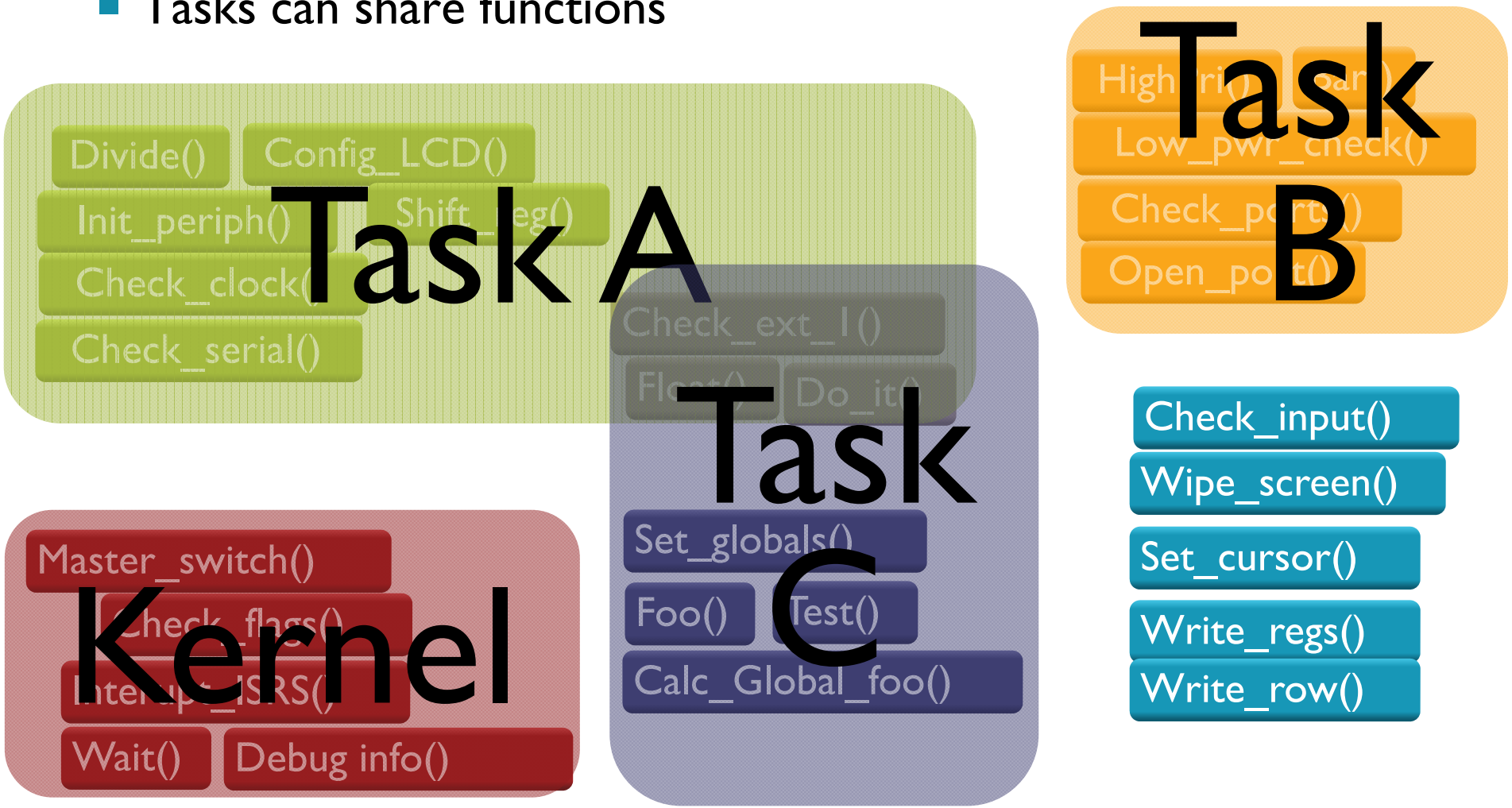
- Tasks can share functions



RTOS -Tasks

Logical Groups can be managed by an RTOS task

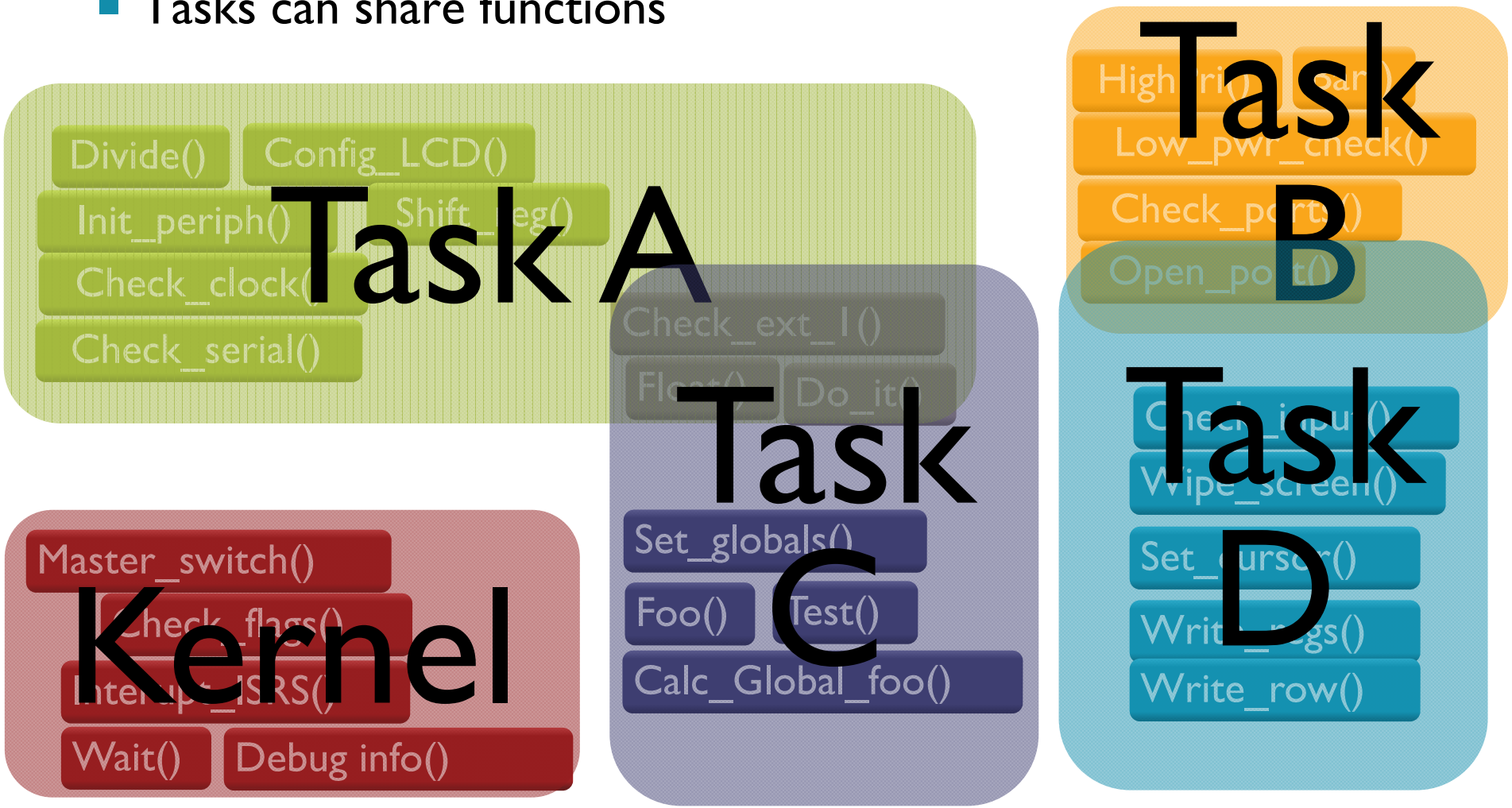
- Tasks can share functions



RTOS -Tasks

Logical Groups can be managed by an RTOS task

- Tasks can share functions



Creating Tasks

```
__task void taskA (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskB (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskC (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskD (void)  {
        While(1)        {
        …
        }
}
```

Tasks are logical components of your project

Tasks do not replace functions

- They call functions

Round Robin Basic Example

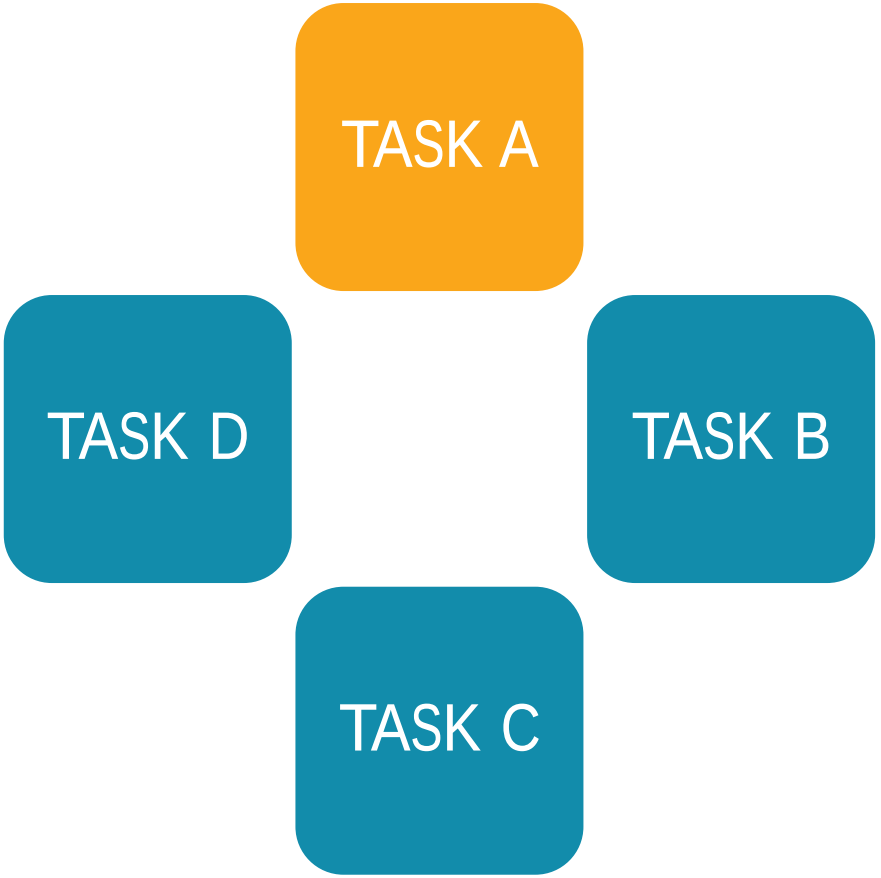
Simplest version of RTOS

- We will go over a more complex version later

Each task gets a time slice in which to run

- Default time slice duration is 5 timer Ticks

Short time slices will make it 'appear' that all tasks execute simultaneously

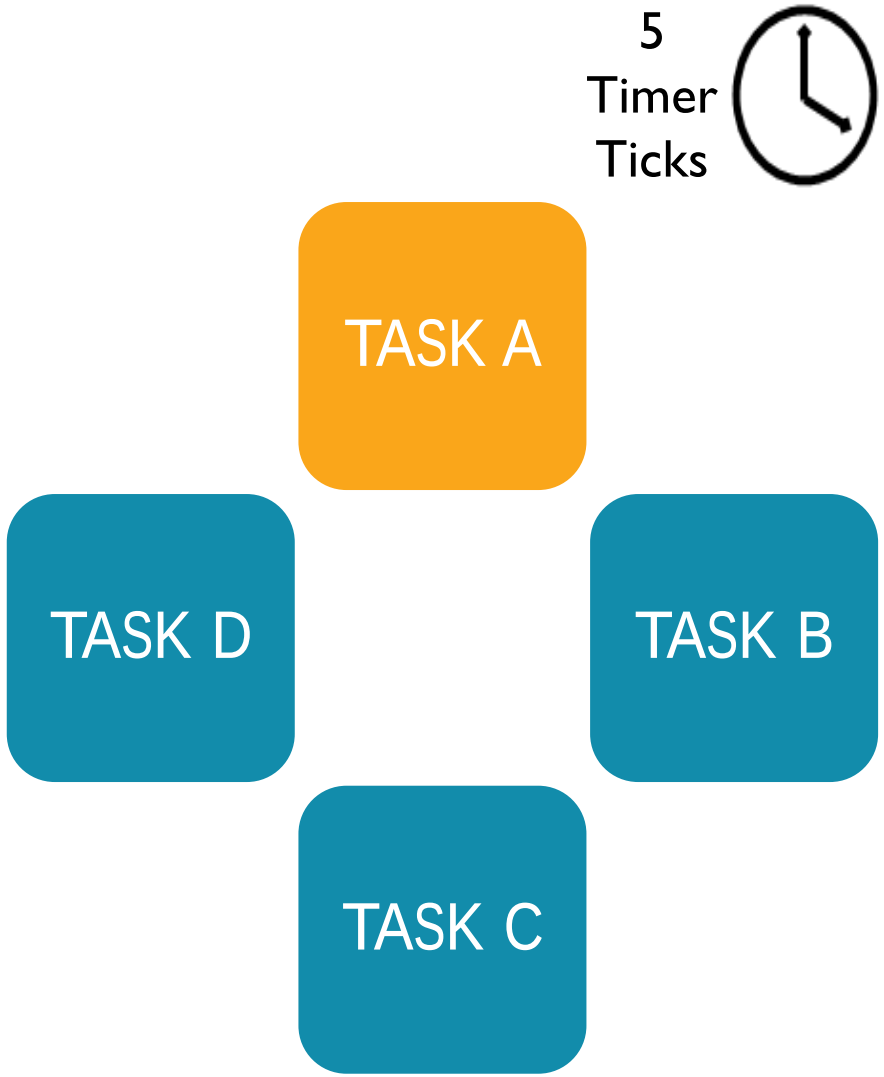


Round Robin Basic Example

Simplest version of RTOS
- We will go over a more complex version later

Each task gets a time slice in which to run
- Default time slice duration is 5 timer Ticks

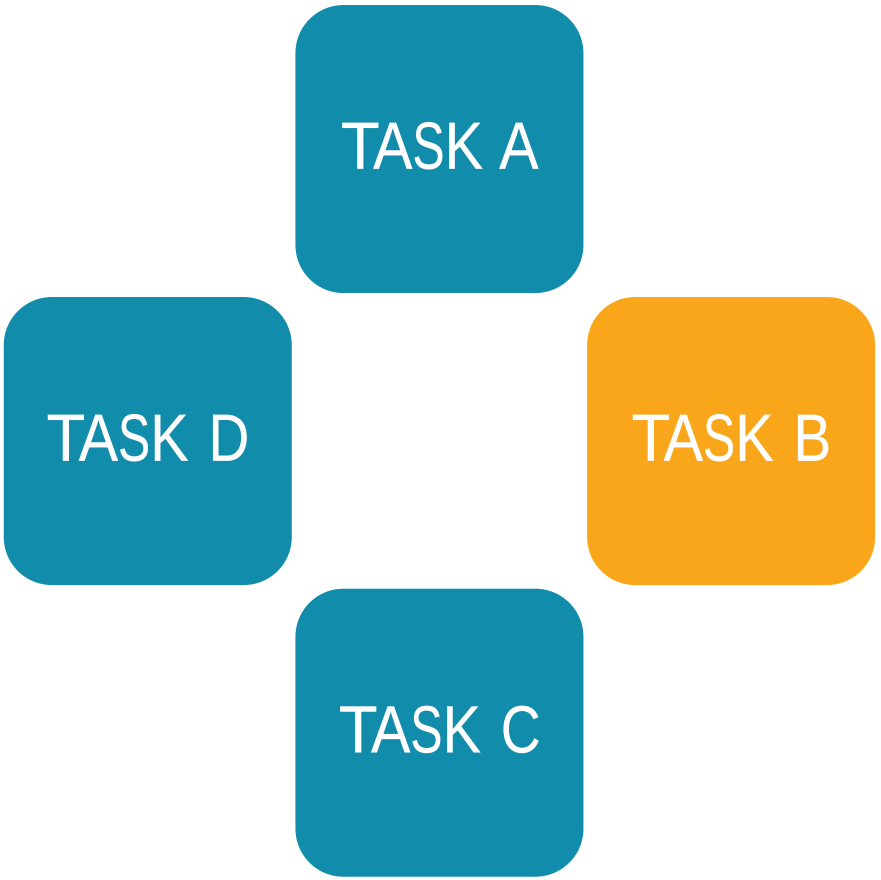
Short time slices will make it 'appear' that all tasks execute simultaneously



Round Robin Basic Example

Timer tick is basic unit of measurement for all RTOS

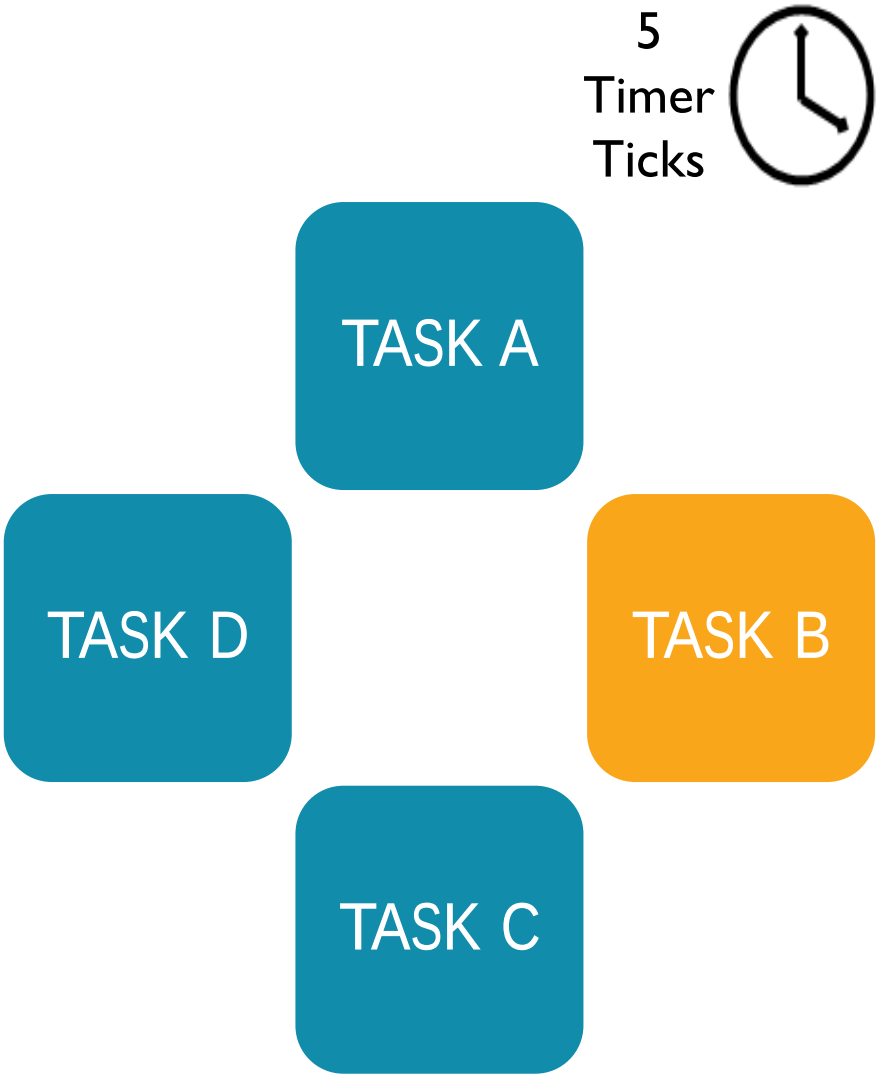
- Basis of all delays, functions, timers, time slices etc.
- Default is 1 timer tick = 10000 timer overflows (10 ms)



Round Robin Basic Example

Timer tick is basic unit of measurement for all RTOS

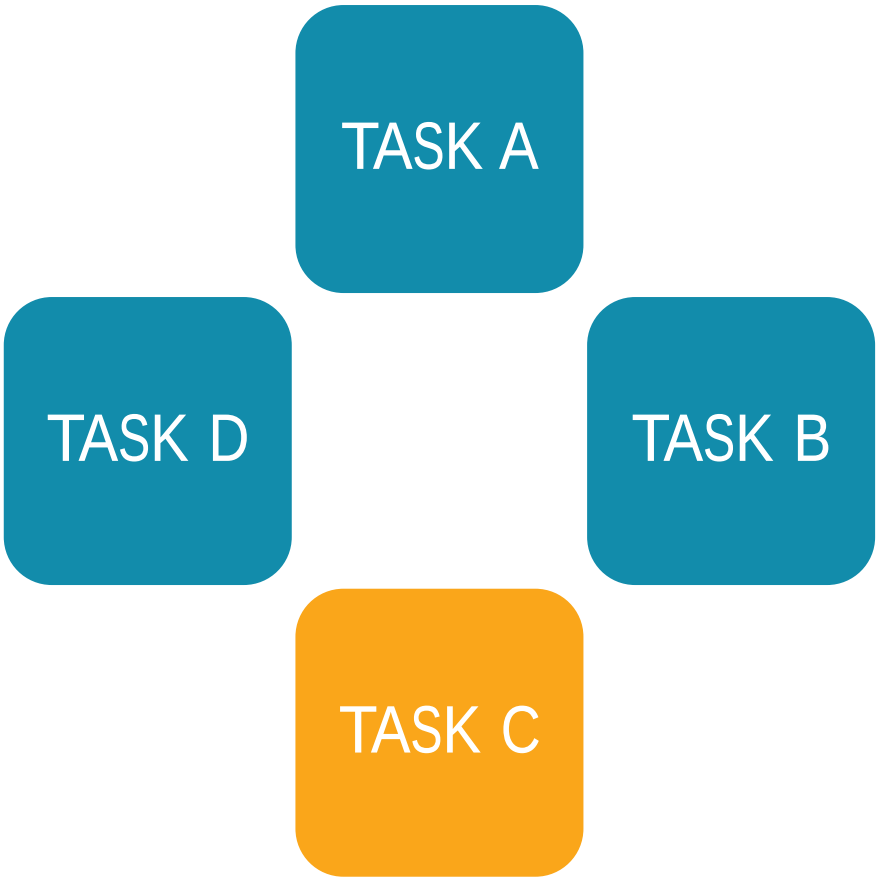
- Basis of all delays, functions, timers, time slices etc.
- Default is 1 timer tick = 10000 timer overflows (10 ms)



Round Robin Basic Example

A task can stop executing before it's time-slice is up RTX switches to the next task that is ready to run if one of these functions are called:

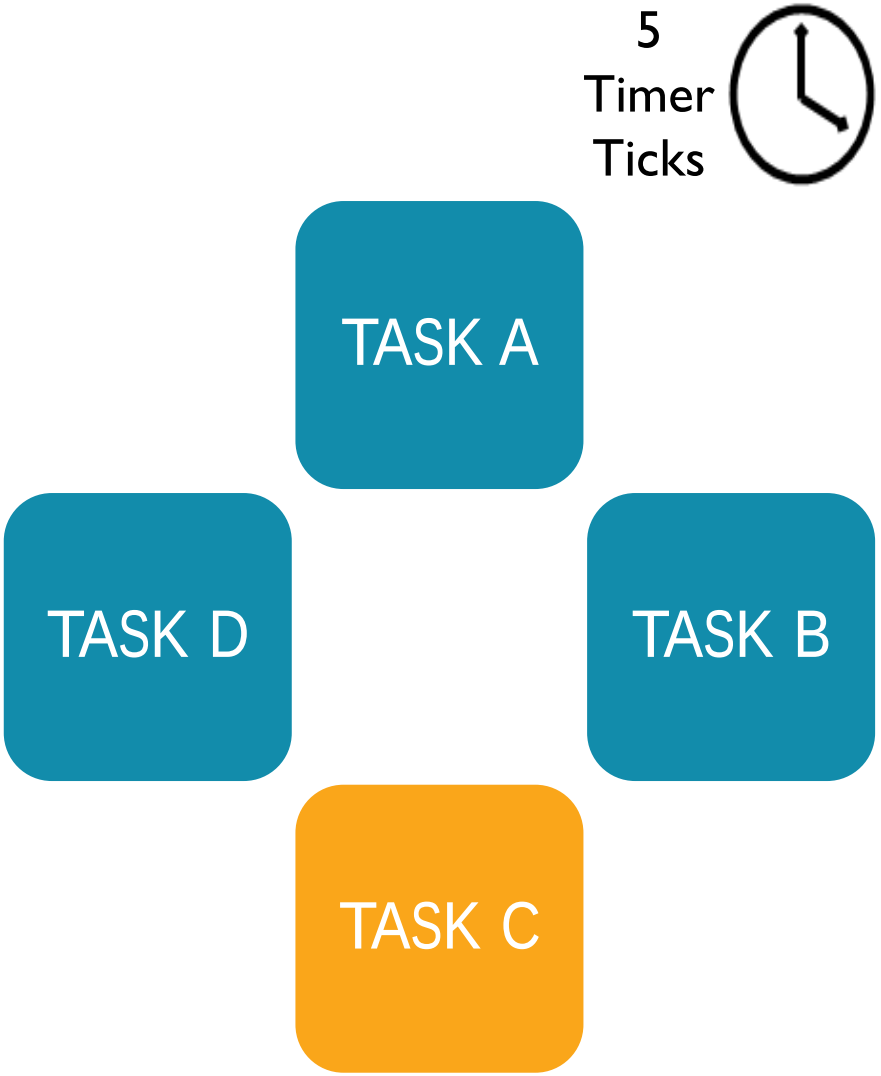
- os_tsk_pass()
- any os_wait library function



Round Robin Basic Example

A task can stop executing before it's time-slice is up
RTX switches to the next task that is ready to run if one of these functions are called:

- os_tsk_pass()
- any os_wait library function



Forcing a Task Switch

```
__task void taskA (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskB (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskC (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskD (void)  {
        While(1)        {
        …
        }
}
```

Adding os_dly_wait() will force taskD to switch earlier than the Round Robin timeout

Forcing a Task Switch

```
__task void taskA (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskB (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskC (void)  {
        While(1)        {
        …
        }
}
```

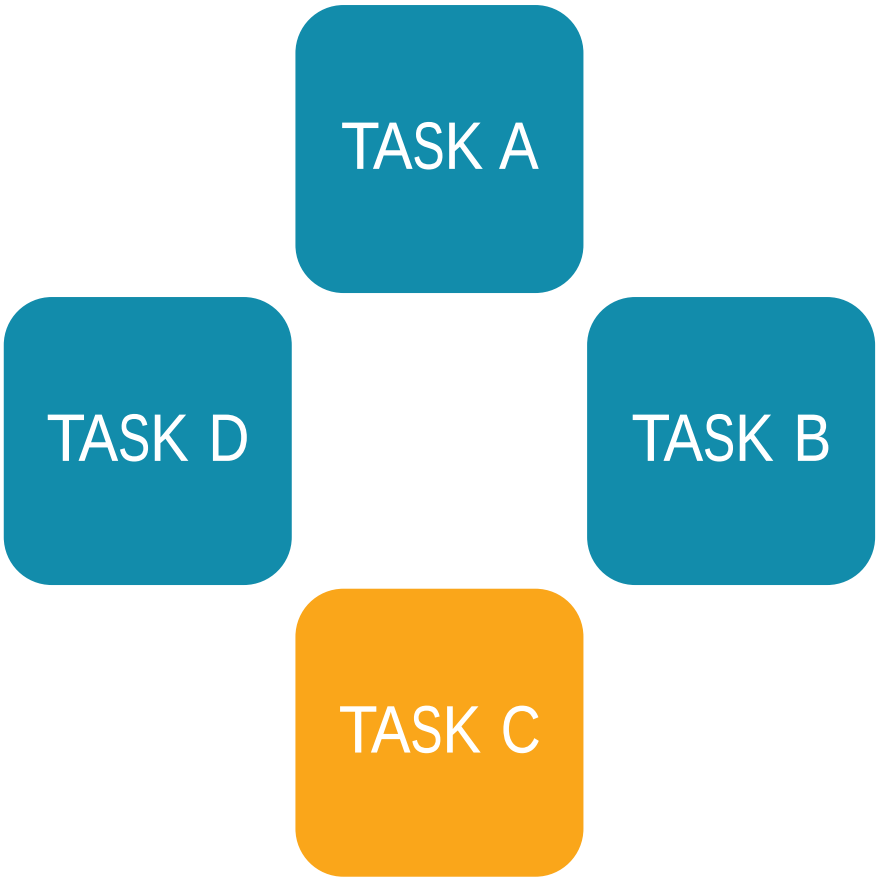
```
__task void taskD (void)  {
        While(1)        {
              os_dly_wait(20);
        }
}
```

Adding os_dly_wait() will force taskD to switch earlier than the Round Robin timeout

Round Robin Basic Example

Task D is running

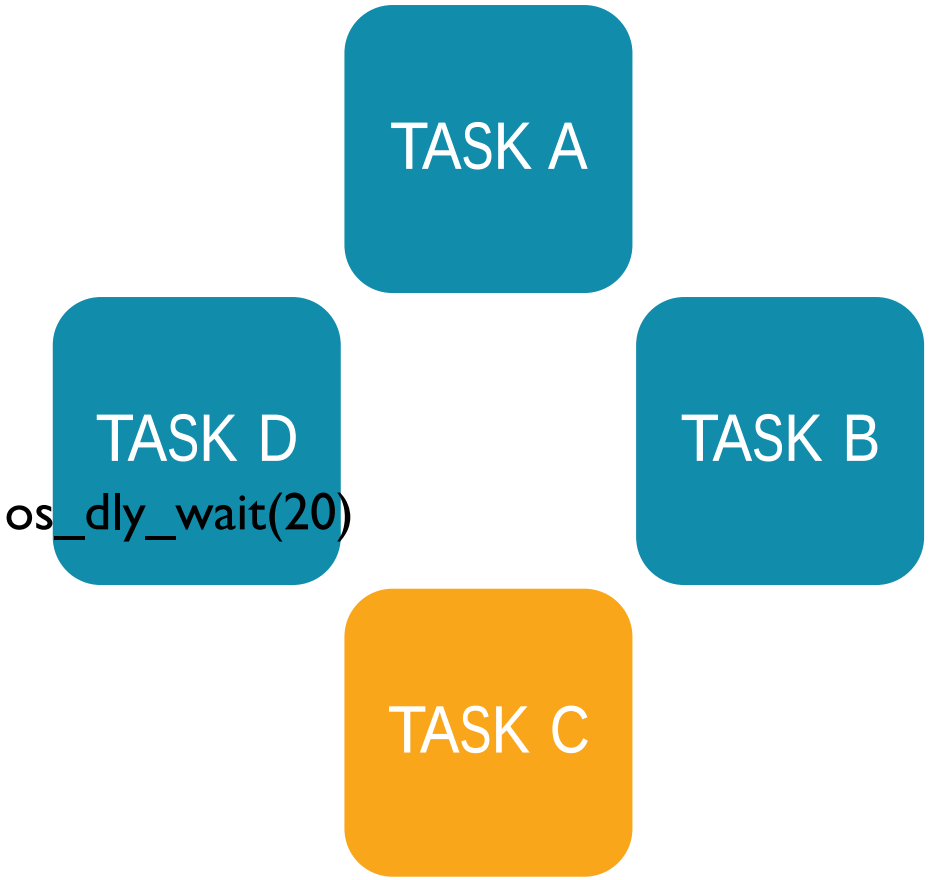
- As soon as wait function is reached, task switch occurs
- Task switches out even though there were 4 more timer ticks to go
- Task D will not be set to a 'READY' state until 20 timer ticks have passed



Round Robin Basic Example

Task D is running

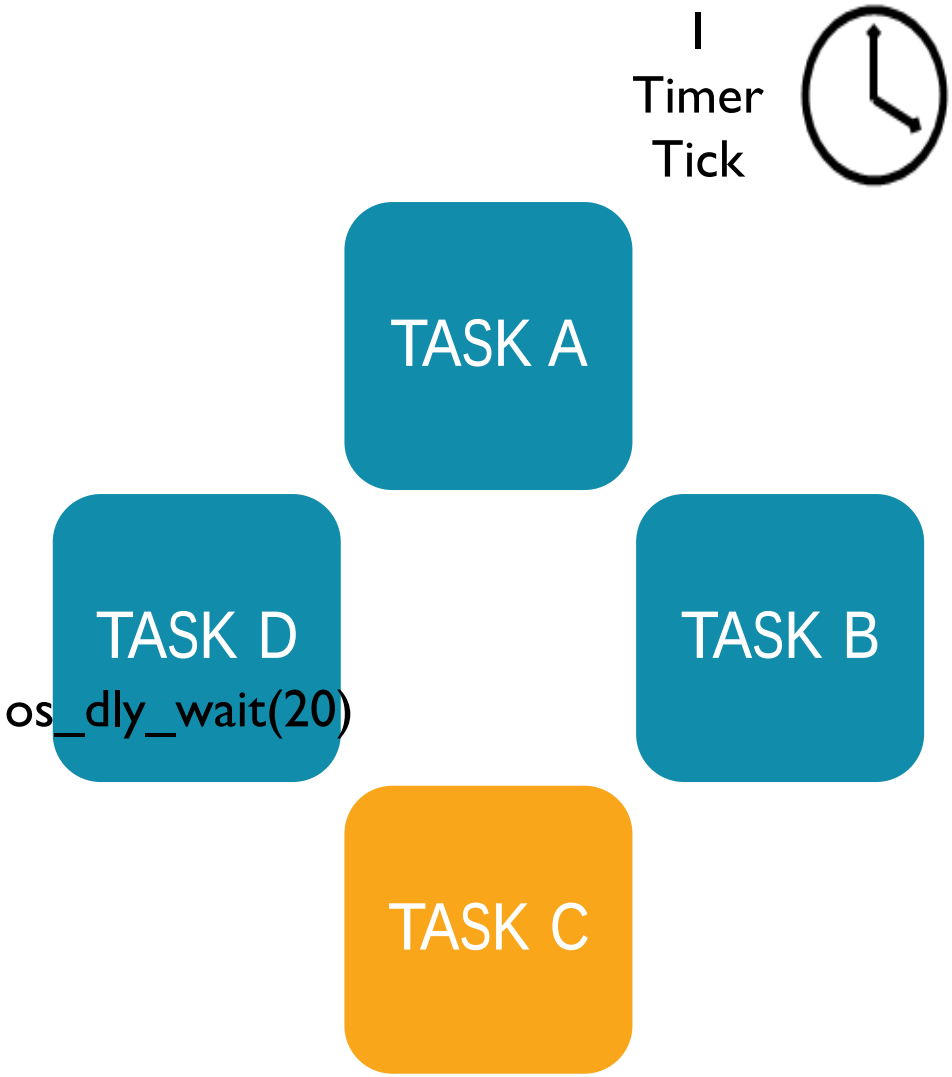
- As soon as wait function is reached, task switch occurs
- Task switches out even though there were 4 more timer ticks to go
- Task D will not be set to a 'READY' state until 20 timer ticks have passed



Round Robin Basic Example

Task D is running

- As soon as wait function is reached, task switch occurs
- Task switches out even though there were 4 more timer ticks to go
- Task D will not be set to a 'READY' state until 20 timer ticks have passed



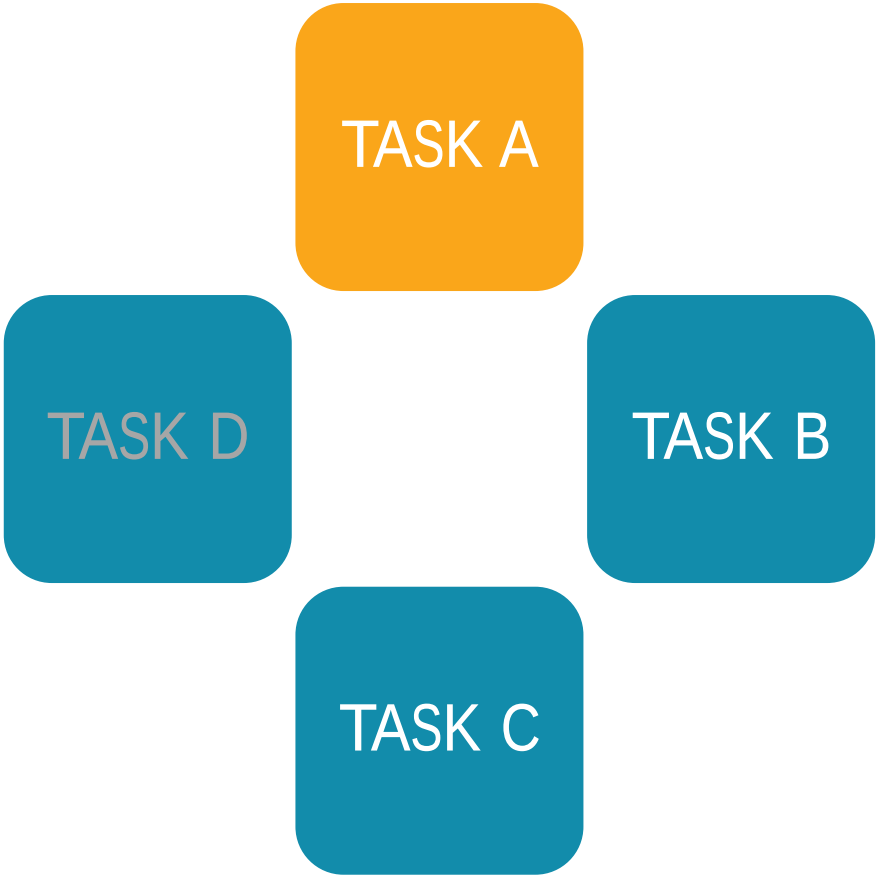
Round Robin Basic Example

Task A task state is 'RUNNING'

Task D task state is 'WAIT_DLY'

The other tasks states are 'READY'

A task set as state 'INACTIVE' will be skipped



RTX Task States

Controlling the Flow of Your Code

```
__task void taskA (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskB (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskC (void)  {
        While(1)        {
        …
        }
}
```

```
__task void taskD (void)  {
        While(1)        {
              os_dly_wait(20);
        }
}
```

- Use ‘waits’ and ‘events’ to control your code, instead of relying on a Round robin Task switch
- More efficient use of your resources

Controlling the Flow of Your Code

```
__task void taskA (void)  {
        While (1)        {
           counterA++;
              os_dly_wait (5);
```

```
__task void taskB (void)  {
        While (1)        {
           counterB++;
           if (counterB == 0) {
              os_tsk_pass ();
```

```
__task void taskC (void)  {
        While(1)        {
           os_evt_wait_or (0x0001, 0xffff);
           counterC++;
```

```
__task void taskD (void)  {
        While(1)        {
              os_dly_wait(20);
        }
}
```

- Use ‘waits’ and ‘events’ to control your code, instead of relying on a Round robin Task switch
- More efficient use of your resources

Creating a RTX Example

RTX is quick to set up, easy to configure

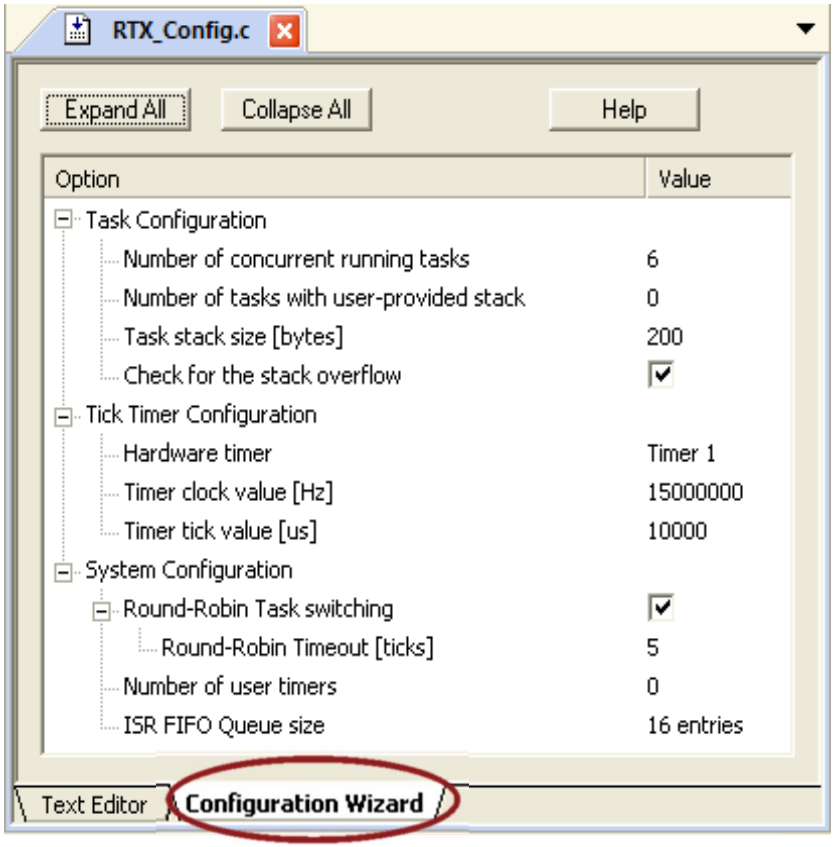
Open the RTX Example

Open:

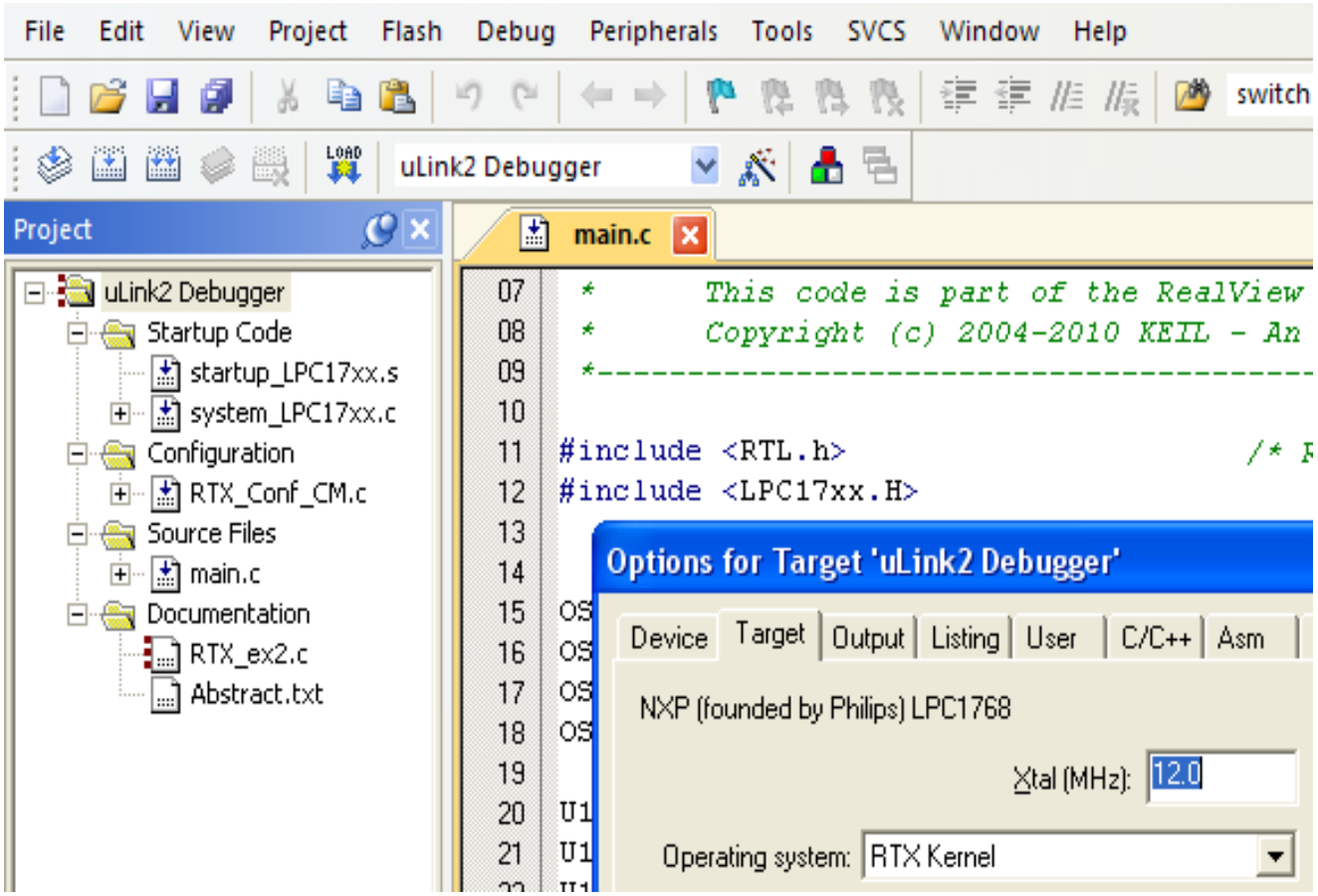
C:\Keil\ARM\RL\RTX\Examples\RTX_ex2\RTX_ex2.uvproj

Example Objectives:

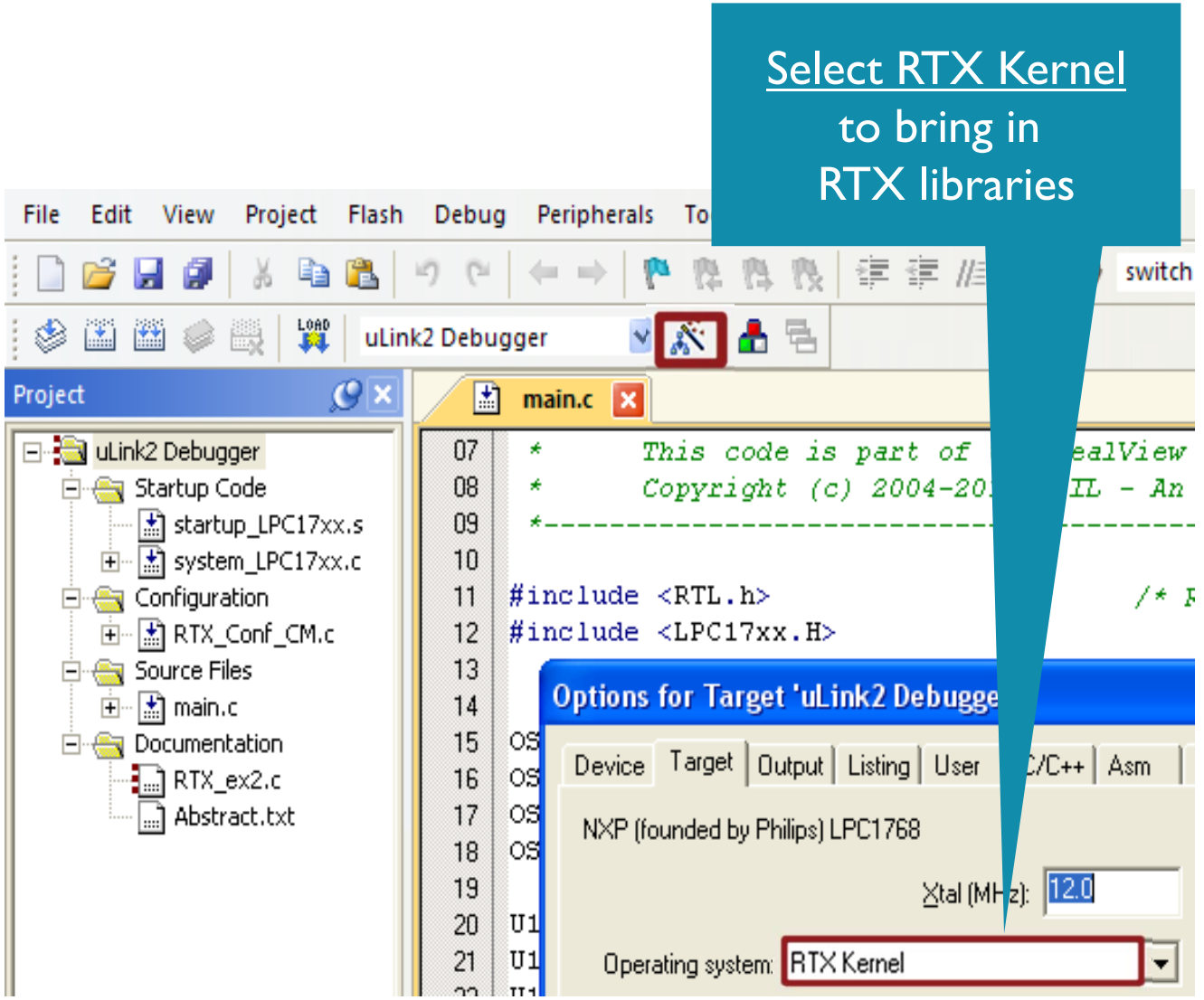
- Configure the RTOS
- Run the program
- Watch the task switch
- Understand RTOS Triggers



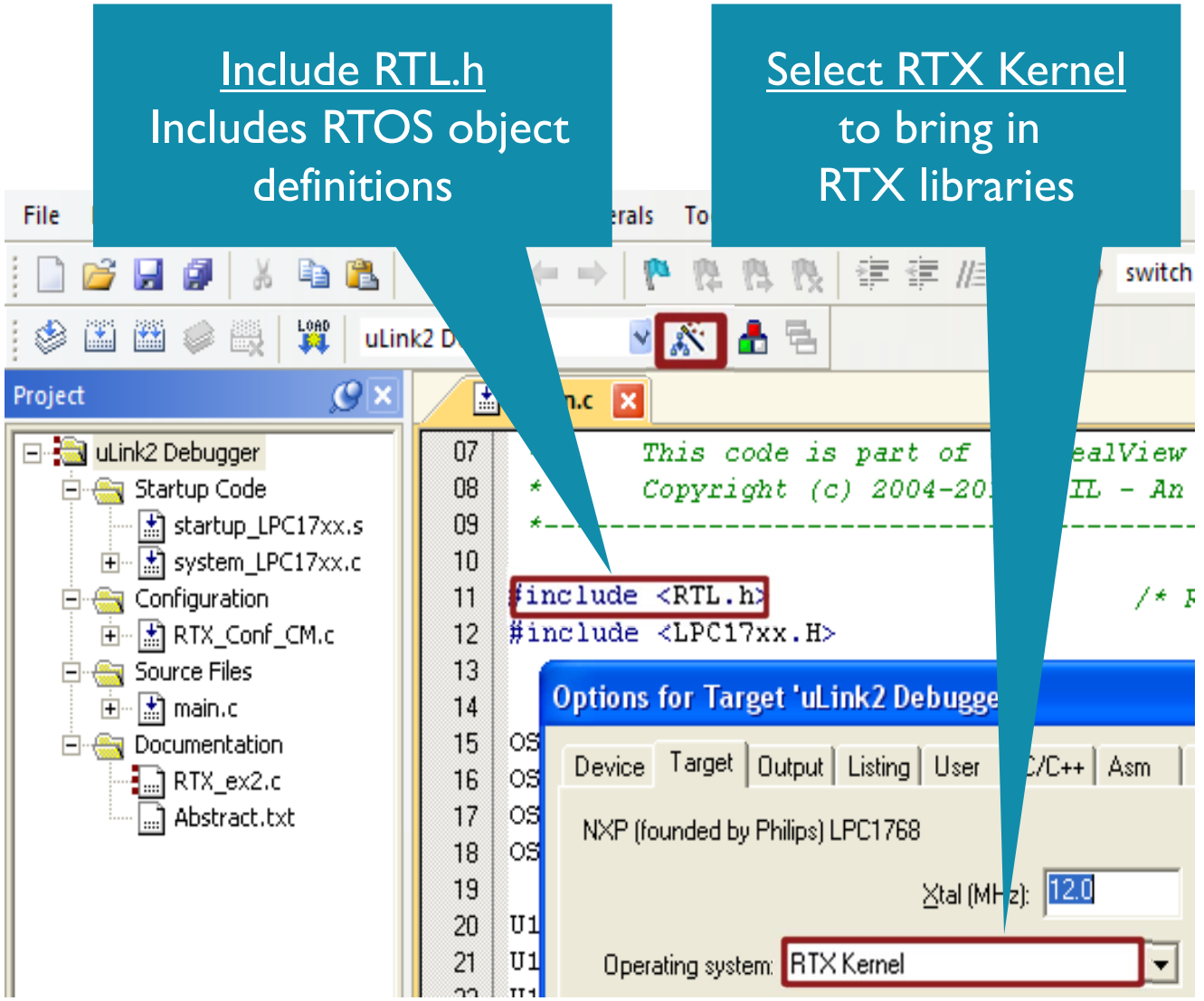
Adding RTX to Your Project



Adding RTX to Your Project



Adding RTX to Your Project

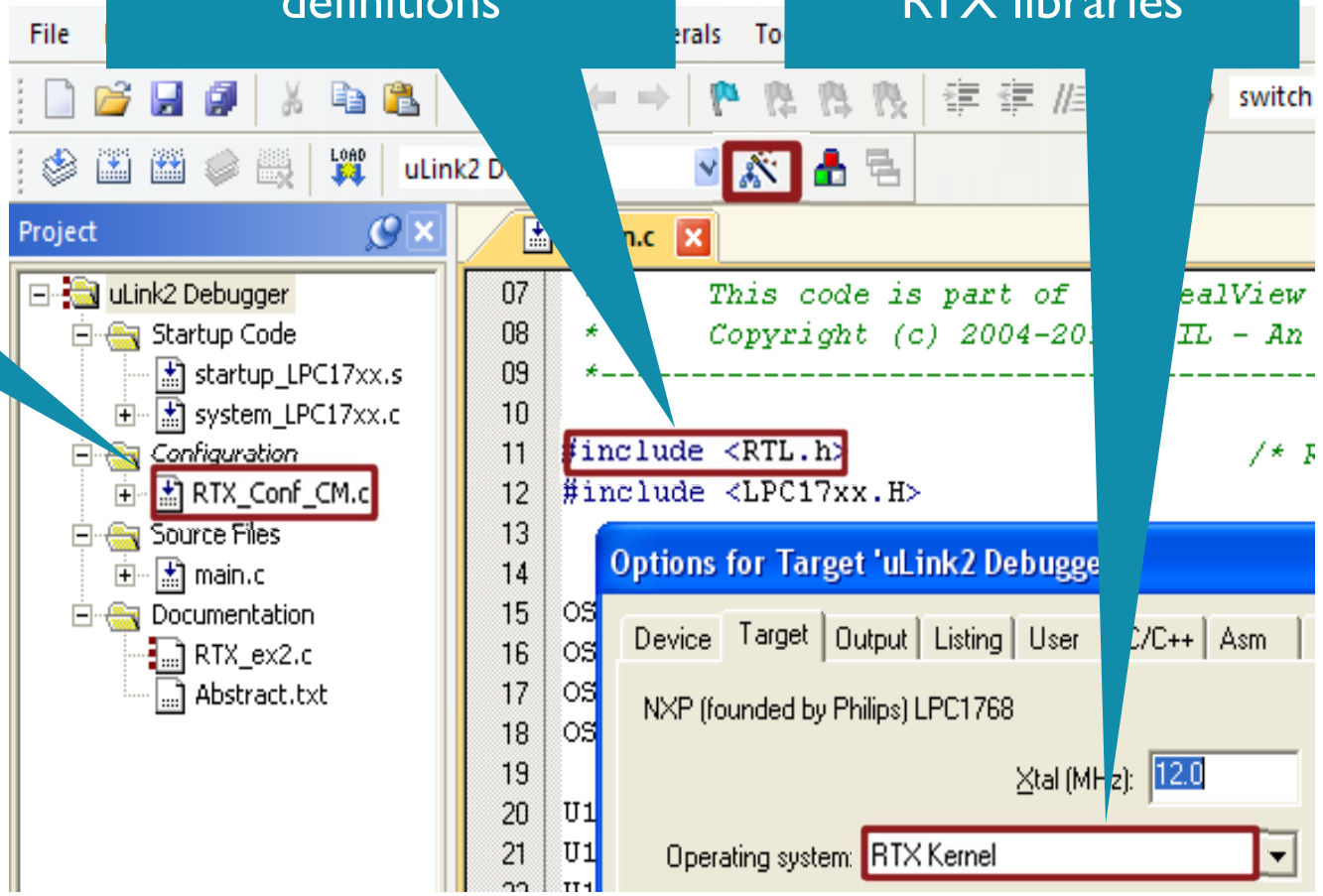


Adding RTX to Your Project

RTX_Conf_CM.c
Configures:
Timer Ticks,
Round Robin,
Tasks

Include RTL.h
Includes RTOS object definitions

Select RTX Kernel
to bring in
RTX libraries



Adding Tasks to your Project

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
  tskA = os_tsk_self ();
  tskB = os_tsk_create (TaskB,1);
  tskC = os_tsk_create (TaskC,1);
  tskD = os_tsk_create (TaskD,2);

  while (1) {
    counterA++;
    os_dly_wait (5);
  }
}
__task void TaskB (void) {
  while (1)  {
    counterB++;
    if (counterB == 0) {
        os_evt_set (0x0001,tskC);
        os_tsk_pass ();
    }
  }
}
__task void TaskC (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counterC++;
  }
}
__task void TaskD (void) {
```

Adding Tasks to your Project

Tasks A-D are using:
- Wait commands
- Set commands
(just like in the example)

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
  tskA = os_tsk_self ();
  tskB = os_tsk_create (TaskB,1);
  tskC = os_tsk_create (TaskC,1);
  tskD = os_tsk_create (TaskD,2);

  while (1) {
    counterA++;
    os_dly_wait (5);
  }
}
__task void TaskB (void) {
  while (1)  {
    counterB++;
    if (counterB == 0) {
        os_evt_set (0x0001,tskC);
        os_tsk_pass ();
    }
  }
}
__task void TaskC (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counterC++;
  }
}
__task void TaskD (void) {
```

Adding Tasks to your Project

Task A is special;
it has to create the other tasks
(more on this later)

Tasks A-D are using:
- Wait commands
- Set commands
(just like in the example)

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
  tskA = os_tsk_self ();
  tskB = os_tsk_create (TaskB,1);
  tskC = os_tsk_create (TaskC,1);
  tskD = os_tsk_create (TaskD,2);

  while (1) {
    counterA++;
    os_dly_wait (5);
  }
}
__task void TaskB (void) {
  while (1)  {
    counterB++;
    if (counterB == 0) {
        os_evt_set (0x0001,tskC);
        os_tsk_pass ();
    }
  }
}
__task void TaskC (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counterC++;
  }
}
__task void TaskD (void) {
```

Adding Tasks to your Project

Task Prototypes

Task A is special;
it has to create the other tasks
(more on this later)

Tasks A-D are using:
- Wait commands
- Set commands
(just like in the example)

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
  tskA = os_tsk_self ();
  tskB = os_tsk_create (TaskB,1);
  tskC = os_tsk_create (TaskC,1);
  tskD = os_tsk_create (TaskD,2);

  while (1) {
    counterA++;
    os_dly_wait (5);
  }
}
__task void TaskB (void) {
  while (1)  {
    counterB++;
    if (counterB == 0) {
        os_evt_set (0x0001,tskC);
        os_tsk_pass ();
    }
  }
}
__task void TaskC (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counterC++;
  }
}
__task void TaskD (void) {
```

Adding Tasks to your Project

Task ID definitions
(Used for setting events,
passing values, etc.)

Task Prototypes

Task A is special;
it has to create the other tasks
(more on this later)

Tasks A-D are using:
- Wait commands
- Set commands
(just like in the example)

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
  tskA = os_tsk_self ();
  tskB = os_tsk_create (TaskB,1);
  tskC = os_tsk_create (TaskC,1);
  tskD = os_tsk_create (TaskD,2);

  while (1) {
    counterA++;
    os_dly_wait (5);
  }
}
__task void TaskB (void) {
  while (1)  {
    counterB++;
    if (counterB == 0) {
        os_evt_set (0x0001,tskC);
        os_tsk_pass ();
    }
  }
}
__task void TaskC (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counterC++;
  }
}
__task void TaskD (void) {
```

Starting up the RTOS in your Code

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the RTOS in your Code

Program must have a main function,
and Main must eventually call:
os_sys_init (XXX);
Where 'XXX' is the first task
to start running

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the RTOS in your Code

Program must have a main function,
and Main must eventually call:
os_sys_init (XXX);
Where 'XXX' is the first task
to start running

Calling:
os_sys_init (TaskA);
Launches RTX kernel and tells it to start
in TaskA

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the Other Tasks

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the Other Tasks

The program jumps to Task A & starts executing

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the Other Tasks

The program jumps to Task A & starts executing

tskA = os_tsk_self ()
Assigns TaskA's ID value to 'tskA'

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the Other Tasks

The program jumps to Task A & starts executing

tskA = os_tsk_self ()
Assigns TaskA's ID value to 'tskA'

tskD = os_tsk_create (TaskD,2);

-Creates TaskD
-Sets it to a priority of 2
-Assigns TaskD's ID value to 'tskD'

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

Starting up the Other Tasks

The program jumps to Task A & starts executing

tskA = os_tsk_self ()
Assigns TaskA's ID value to 'tskA'

tskD = os_tsk_create (TaskD,2);

-Creates TaskD
-Sets it to a priority of 2
-Assigns TaskD's ID value to 'tskD'

Besides creating tasks,
There is the option to delete them

```
OS_TID tskA;
OS_TID tskB;
OS_TID tskC;
OS_TID tskD;

__task void TaskA (void);
__task void TaskB (void);
__task void TaskC (void);
__task void TaskD (void);

__task void TaskA (void) {
tskA = os_tsk_self ();
tskB = os_tsk_create (TaskB,1);
tskC = os_tsk_create (TaskC,1);
tskD = os_tsk_create (TaskD,2);
  while (1) {
    counter1++;
    os_dly_wait (5);
  }
}

. . .


_task void TaskD (void) {
  while (1) {
    os_evt_wait_or (0x0001, 0xffff);
    counter4++;
  }
}


int main (void) {
  os_sys_init (TaskA);
}
```

RTOS Objects

Use Cooperative Multitasking for Greater Efficiency

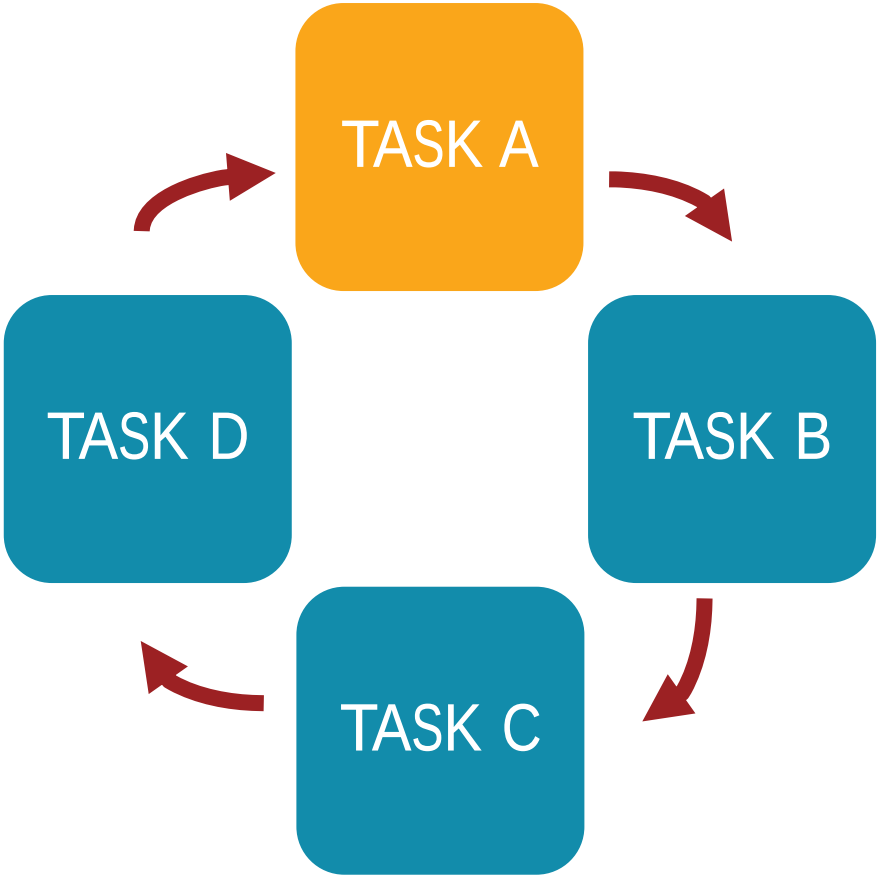
Cooperative Multitasking

Round Robin is not scalable

Cooperative Multitasking gives more flexibility

- The task scheduler processes all the tasks and then puts the highest ready task into the running state
- The highest priority task can then continue with its execution

Use Semaphores and other RTOS objects for inter-task communication



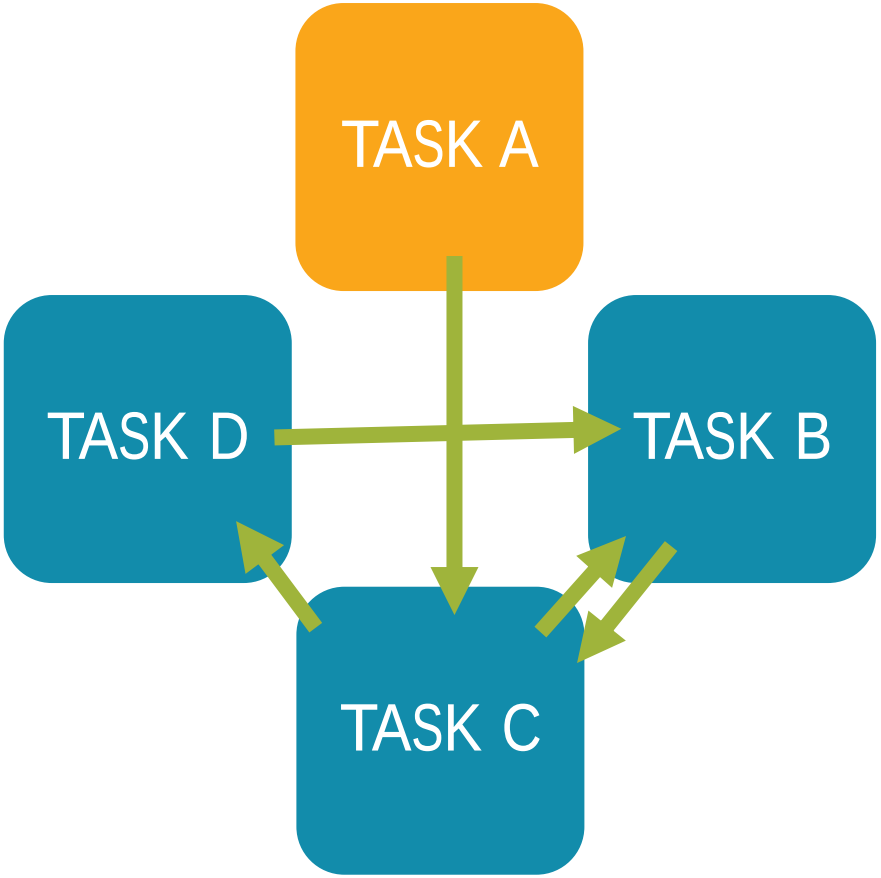
Cooperative Multitasking

Round Robin is not scalable

Cooperative Multitasking gives more flexibility

- The task scheduler processes all the tasks and then puts the highest ready task into the running state
- The highest priority task can then continue with its execution

Use Semaphores and other RTOS objects for inter-task communication



RTX OS wait Library Functions

RTX Library function	Task State
os_dly_wait()	WAIT_DLY
os_itv_wait()	WAIT_ITV
os_evt_wait_or()	WAIT_OR
os_evt_wait_and()	WAIT_AND
os_sem_wait()	WAIT_SEM
os_mut_wait()	WAIT_MUT
os_mbx_wait()	WAIT_MBX

Managing Resources With Semaphores

Round Robin

- Task switch at 1 Tick
- Serial data is garbled

Similar issue can occur with memory access

Need to use a Semaphore

- It is like a virtual token
- Used when a resource is needed by multiple tasks

```
__task void taskA (void)  {
        While(1)        {
        Printf("Oh, sweet Caroline, Good times never seem…");
        }
```

```
__task void taskB (void)  {
        While(1)        {
        Printf("And we're rolling, rolling, rolling down the…");
        }
```

```
__task void taskC (void)  {
        While(1)        {
        Printf("I tell ya, life ain't easy for a boy named Sue");
        }
```

Serial Out

Managing Resources With Semaphores

When there is a free semaphore:

- The first task that requests it gets the token
- No other task can obtain the token until it is released


```
__task void taskA (void)  {
        While(1)          {
        Printf("Oh, sweet Caroline, Good times never seem…");
        }
```

```
__task void taskB (void)  {
        While(1)          {
        Printf("And we're rolling, rolling, rolling down the…");
        }
```

```
__task void taskC (void)  {
        While(1)          {
        Printf("I tell ya, life ain't easy for a boy named Sue");
        }
```

Serial Out

```
Oh, sweet Caroline, Good times never seem…
```

Managing Resources With Semaphores

When a task requests a semaphore that is not available:

- Task is put to sleep

When the token is returned to the semaphore:

- Task wakes up and its status set to ready


```
__task void taskA (void)  {
        While(1)        {
        Printf("Oh, sweet Caroline, Good times never seem…");
        }
```

```
__task void taskB (void)  {
        While(1)        {
        Printf("And we're rolling, rolling, rolling down the…");
        }
```

```
__task void taskC (void)  {
        While(1)        {
        Printf("I tell ya, life ain't easy for a boy named Sue");
        }
```

Serial Out

```
Oh, sweet Caroline, Good times never seem…
And we're rolling, rolling, rolling down the…
```

Sharing a Serial Port With Different Tasks

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Sharing a Serial Port With Different Tasks

Create a semaphore to be used in the example

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Sharing a Serial Port With Different Tasks

Create a semaphore to be used in the example

os_sem_wait
Sees if semaphore1 is free

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Sharing a Serial Port With Different Tasks

Create a semaphore to be used in the example

os_sem_wait
Sees if semaphore1 is free

os_sem_send
Frees up semaphore1

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Sharing a Serial Port With Different Tasks

Create a semaphore to be used in the example

os_sem_wait
Sees if semaphore1 is free

os_sem_send
Frees up semaphore1

Low priority tasks sees if Semaphore1 is free

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Sharing a Serial Port With Different Tasks

Create a semaphore to be used in the example

os_sem_wait
Sees if semaphore1 is free

os_sem_send
Frees up semaphore1

Low priority tasks sees if Semaphore1 is free

Also Supported in RTX: Counting Semaphores (as opposed to binary)

```
OS_SEM semaphore1;

//    Task 1 - High Priority - Active every 3 ticks

__task void task1 (void) {
  OS_RESULT ret;

  while (1) {
    /* Pass control to other tasks for 3 OS ticks */
    os_dly_wait(3);
    /* Wait 1 ticks for the free semaphore */
    ret = os_sem_wait (semaphore1, 1);
    if (ret != OS_R_TMO) {
      /* If there was no time-out the semaphore
          was acquired */
      printf ("Task 1\n");
      /* Return a token back to a semaphore */
      os_sem_send (semaphore1);
    }
  }
}

//    Task 2 - Low Priority * -  looks for a free semaphore

__task void task2 (void) {
  while (1) {
    /* Wait indefinitely for a free semaphore */
    os_sem_wait (semaphore1, 0xFFFF);
    /* OK, the serial interface is free now, use it. */
    printf ("Task 2 \n");
    /* Return a token back to a semaphore. */
    os_sem_send (semaphore1);
  }
}
```

Other RTX Objects

Mutexes and Mailboxes

Mutexes

(Mutual exclusion locks)

- A software object that locks a common resource

Similar to a semaphore

- But any task can release a semaphore token

Only the task that locks the common resource can access it

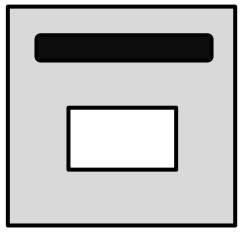
- Big feature – Priority inheritance (more later)

Mailboxes – Pass Messages Between Tasks

A task acquires a premade mailbox and waits for messages

If a message in a specified mailbox is not available:

- The waiting task is put to sleep
- Task wakes up as soon as another task sends a message to the mailbox



```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

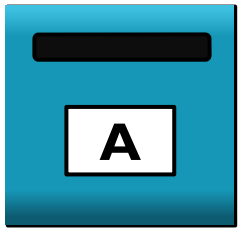
```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

Mailboxes – Pass Messages Between Tasks

A task acquires a premade mailbox and waits for messages

If a message in a specified mailbox is not available:

- The waiting task is put to sleep
- Task wakes up as soon as another task sends a message to the mailbox




```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

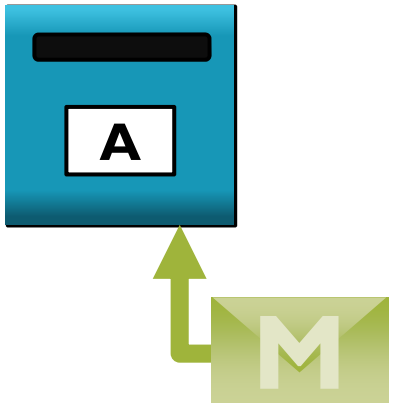
```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

Mailboxes – Pass Messages Between Tasks

A task acquires a premade mailbox and waits for messages

If a message in a specified mailbox is not available:

- The waiting task is put to sleep
- Task wakes up as soon as another task sends a message to the mailbox




```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

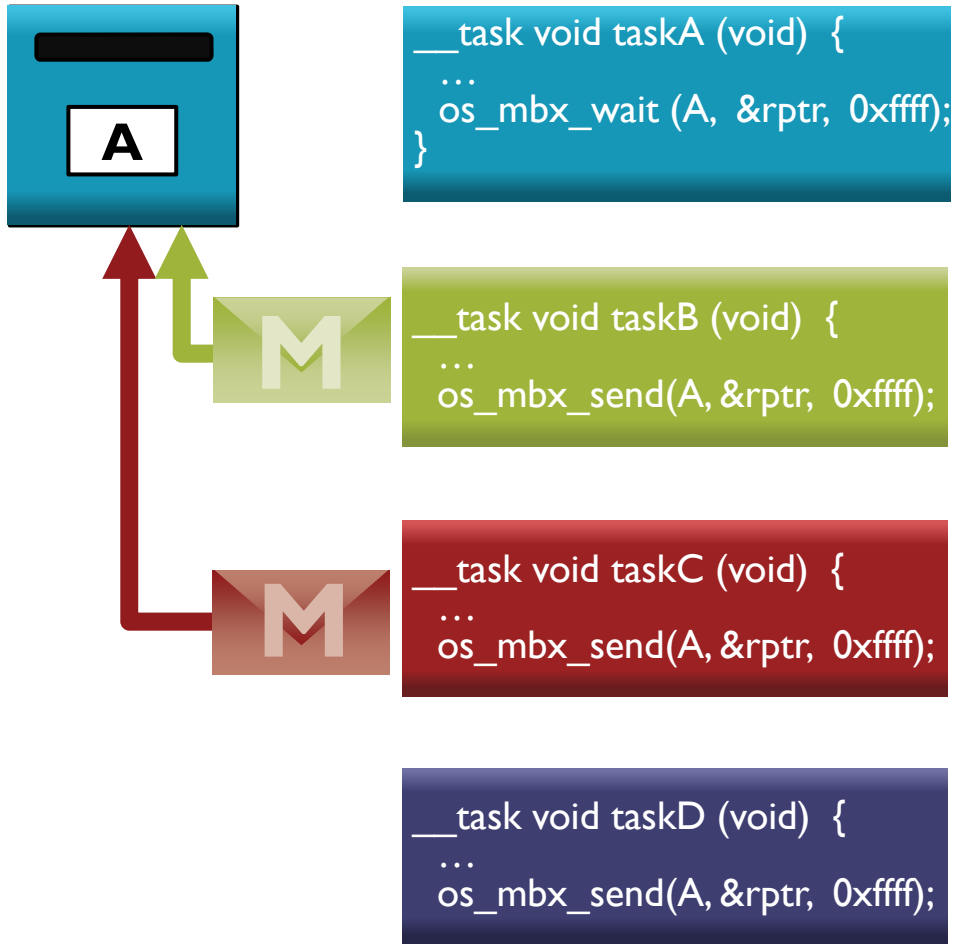
```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

Mailboxes – Pass Messages Between Tasks

A task acquires a premade mailbox and waits for messages

If a message in a specified mailbox is not available:

- The waiting task is put to sleep
- Task wakes up as soon as another task sends a message to the mailbox

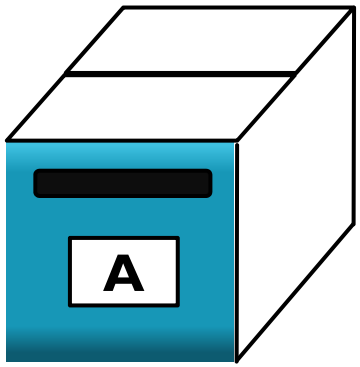


Mailboxes – Pass Messages Between Tasks

When a Mailbox is created it has a fixed size

- If mail box is full, task that wants to send a message is put to sleep until a space is emptied

Dynamic allocation / freeing of memory blocks must be done by programmer (used to prevent memory leaks)




```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

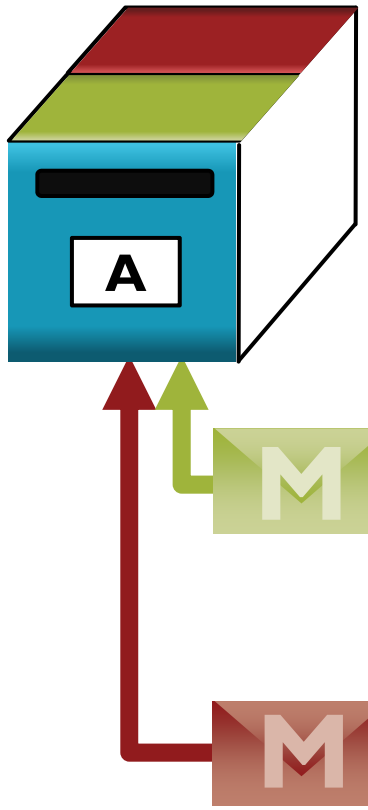
```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

Mailboxes – Pass Messages Between Tasks

When a Mailbox is created it
- has a fixed size
  - If mail box is full, task that wants to send a message is put to sleep until a space is emptied

Dynamic allocation / freeing of
- memory blocks must be done by programmer (used to prevent memory leaks)




```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

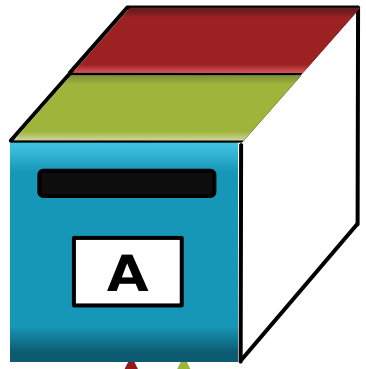
```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

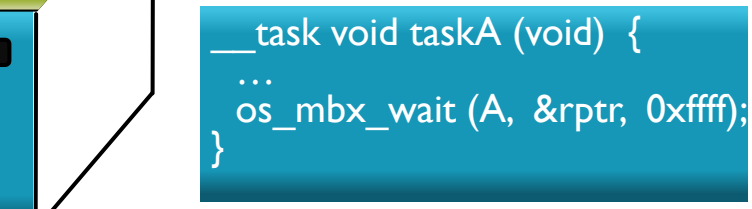
```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

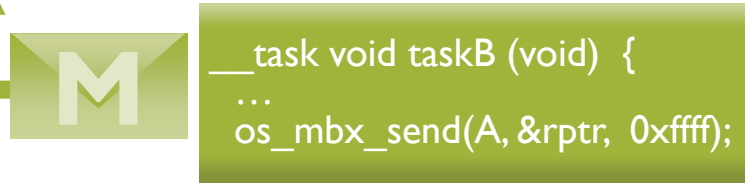
Mailboxes – Pass Messages Between Tasks

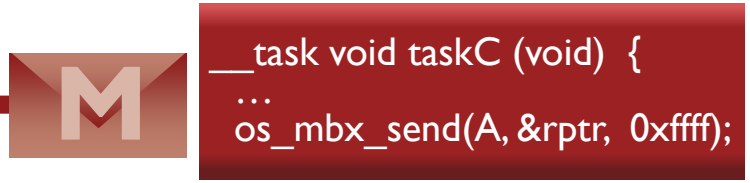
When a Mailbox is created it
- has a fixed size
  - If mail box is full, task that wants to send a message is put to sleep until a space is emptied

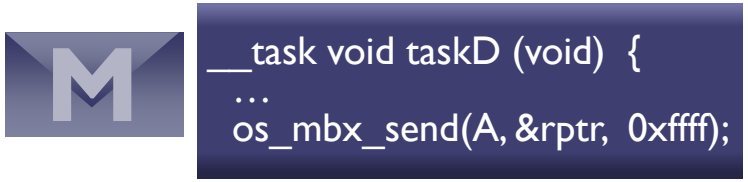
Dynamic allocation / freeing of
- memory blocks must be done by programmer (used to prevent memory leaks)







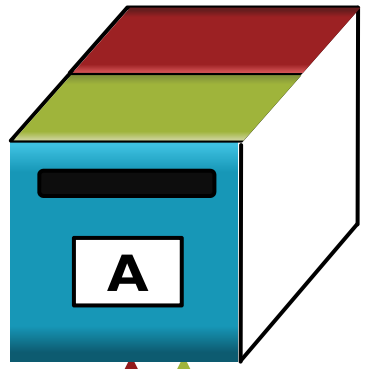




Mailboxes – Pass Messages Between Tasks

When a Mailbox is created it
- has a fixed size
  - If mail box is full, task that wants to send a message is put to sleep until a space is emptied

Dynamic allocation / freeing of
- memory blocks must be done by programmer (used to prevent memory leaks)




```
__task void taskA (void)  {
  …
  os_mbx_wait (A,  &rptr,  0xffff);
}
```

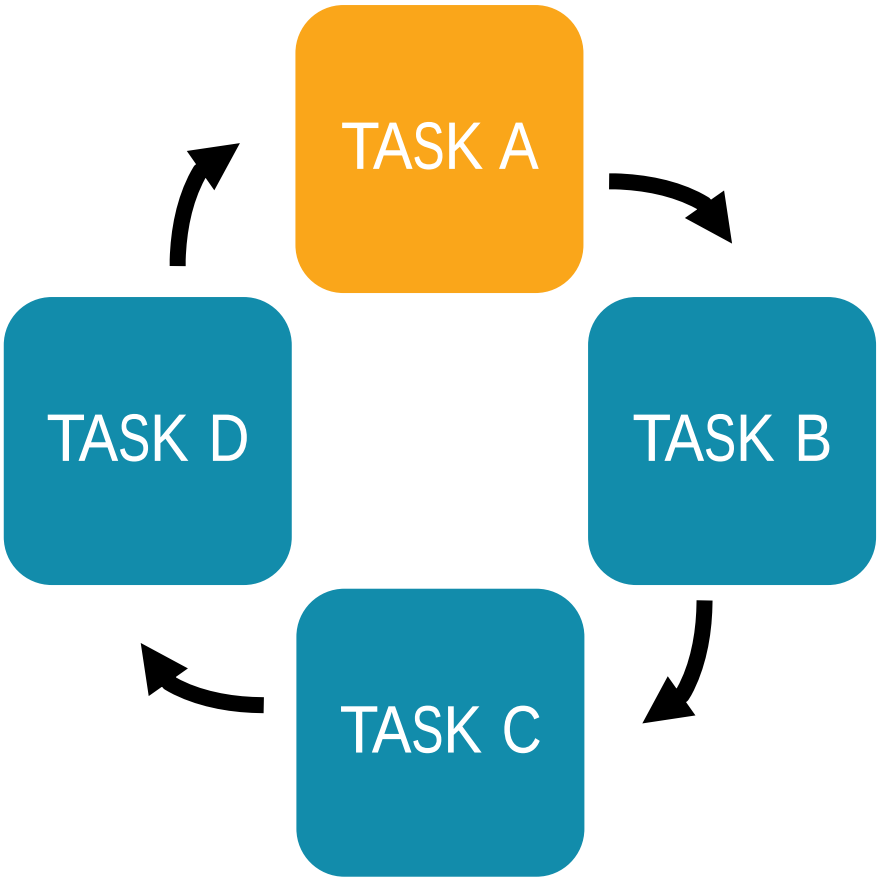
```
__task void taskB (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskC (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

```
__task void taskD (void)  {
  …
  os_mbx_send(A, &rptr,  0xffff);
```

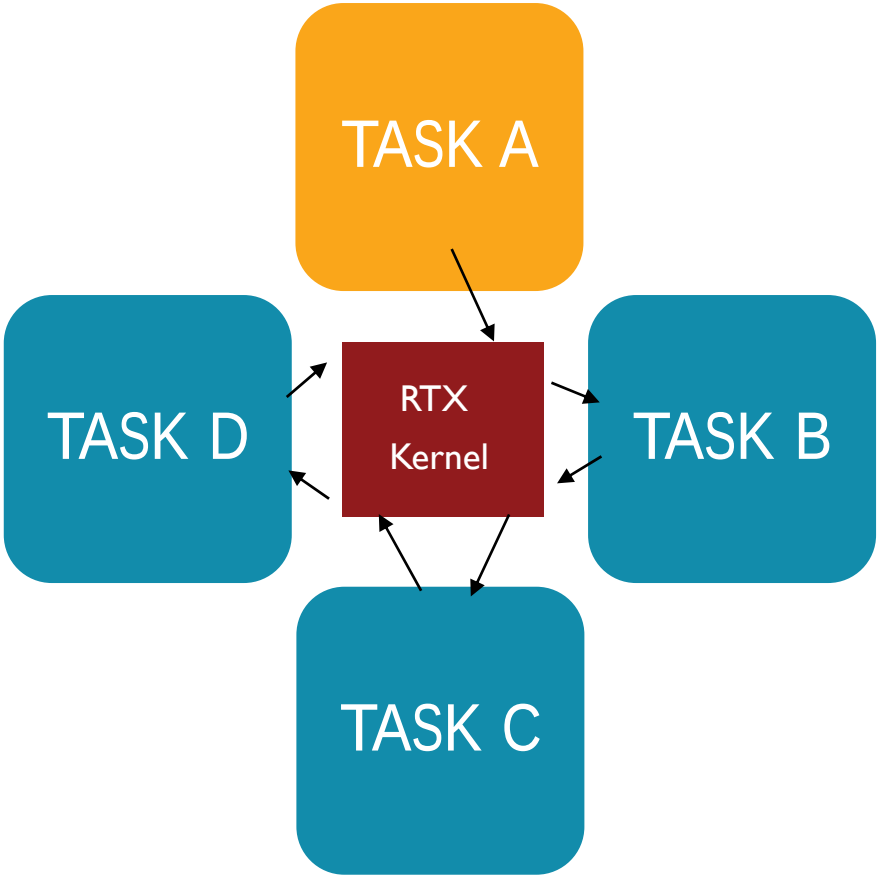
Interrupts are handled by Kernel

To initiate a task switch, the RTX
    Kernel is called
Kernel manages the time plus
    other house keeping chores
Kernel will alert a task if an
    interrupt sends it a message



Interrupts are handled by Kernel

To initiate a task switch, the RTX
    Kernel is called
Kernel manages the time plus
    other house keeping chores
Kernel will alert a task if an
    interrupt sends it a message



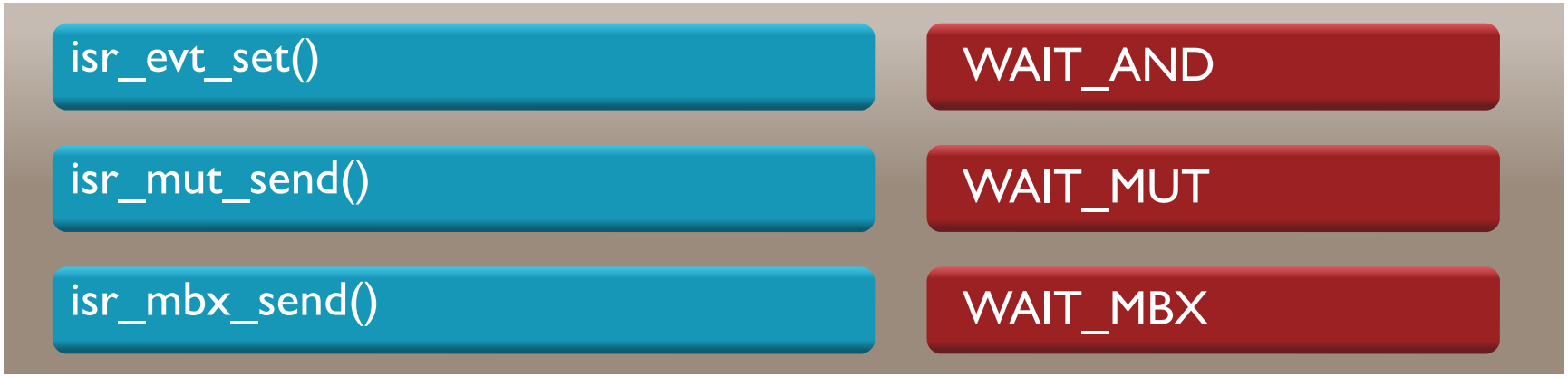
ARM interrupt functions

The ARM interrupt function sets a flag that the task is waiting for

- Synchronizes an external asynchronous event to an RTX kernel task
- Shortens Interrupt Service Routines

On Cortex-M devices, interrupts are never disabled by RTX RTOS

- No overhead on interrupt latency



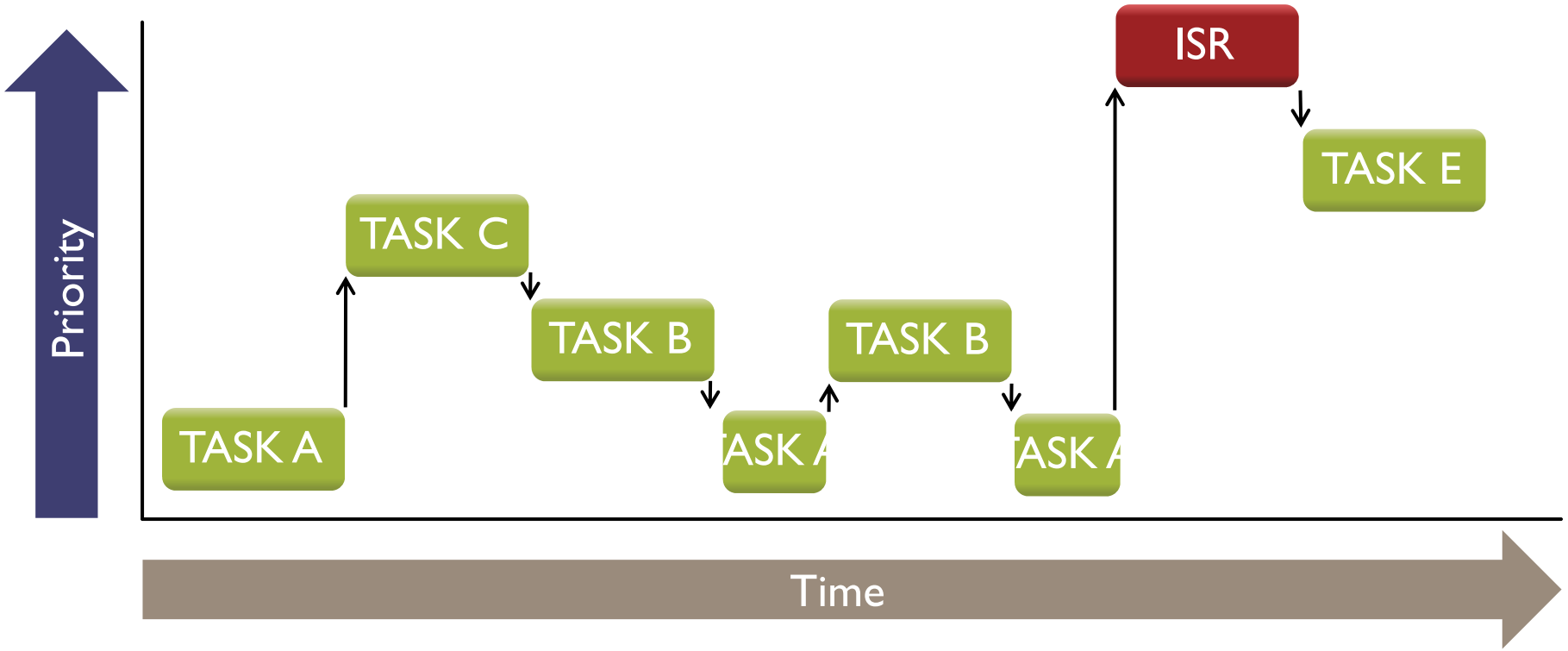
Preemptive Task Switching

Giving Even More Control to Your Project

Preemptive Task Switching

Assign priorities to different tasks for greater control of data flow

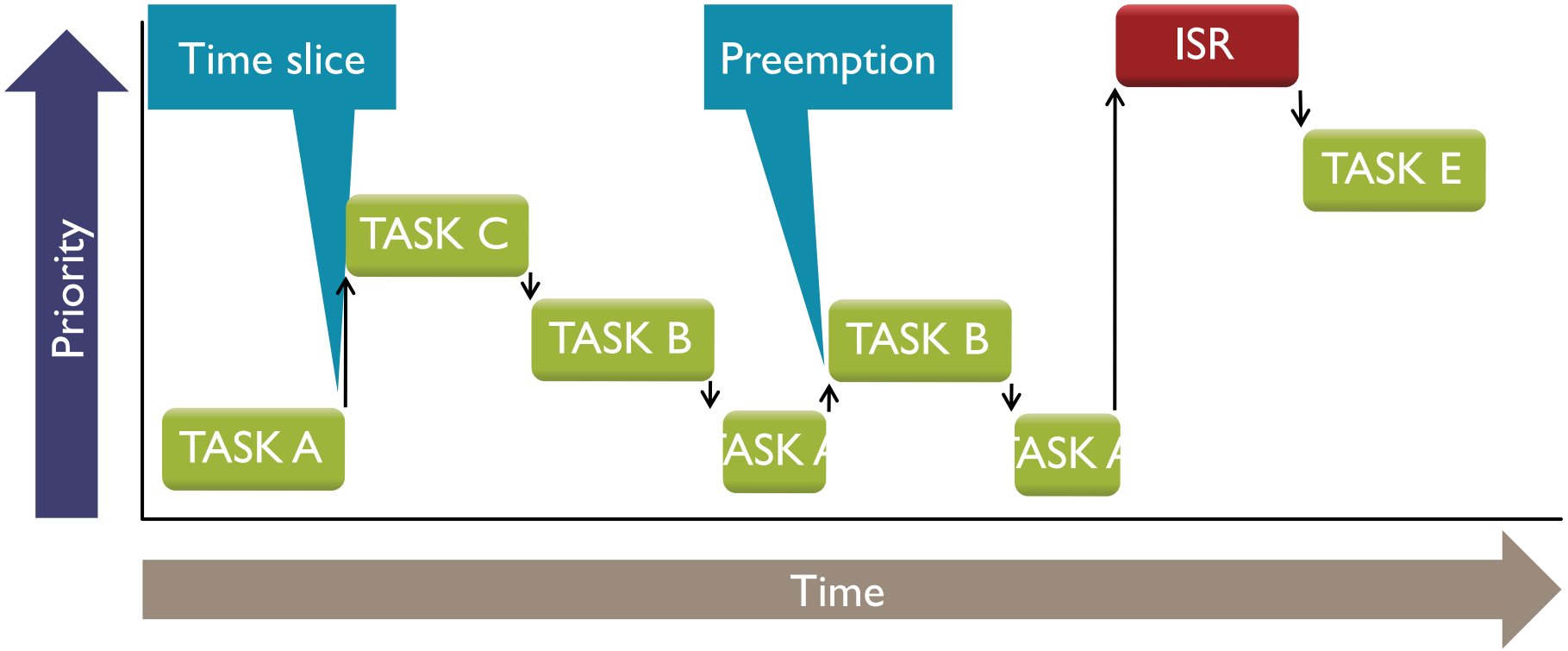
- No need to service a housekeeping message, if the fire alarm is going off



Preemptive Task Switching

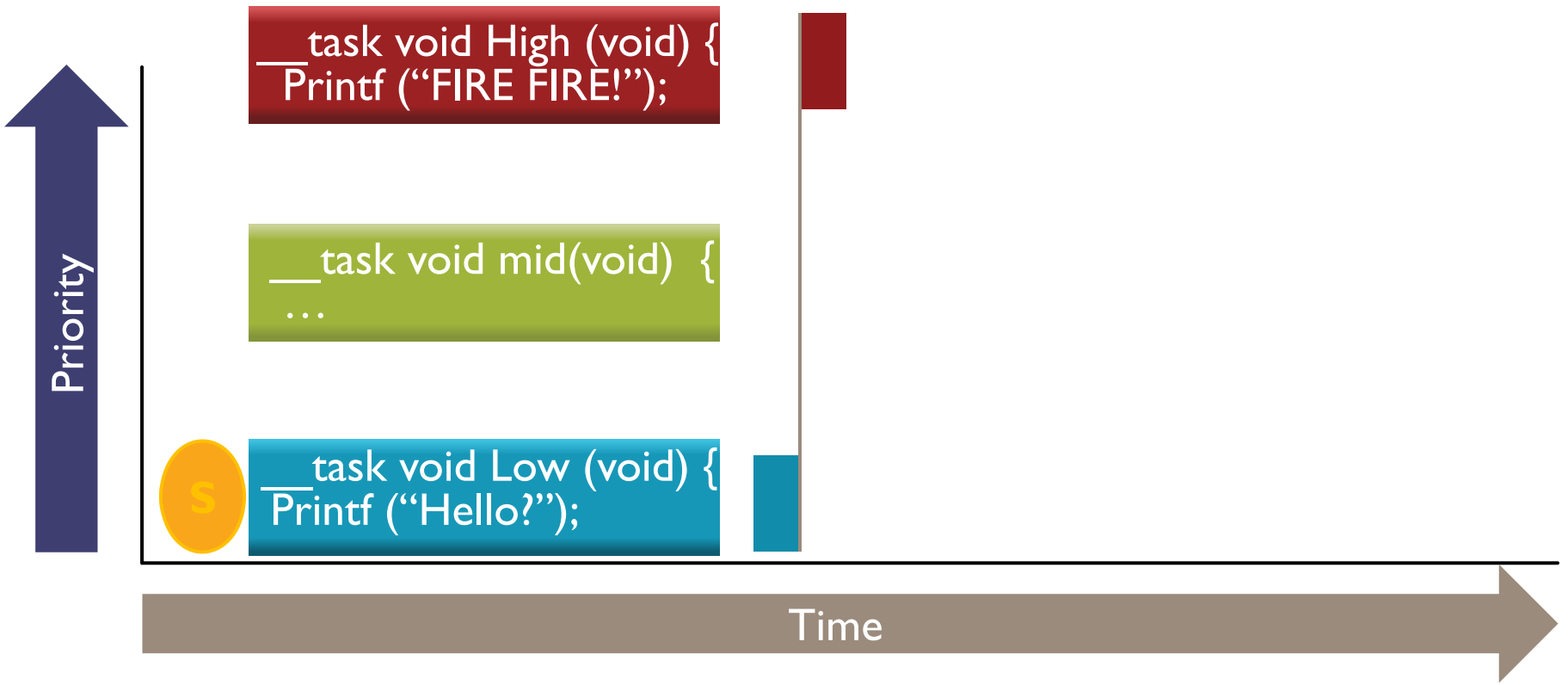
Assign priorities to different tasks for greater control of data flow

- No need to service a housekeeping message, if the fire alarm is going off



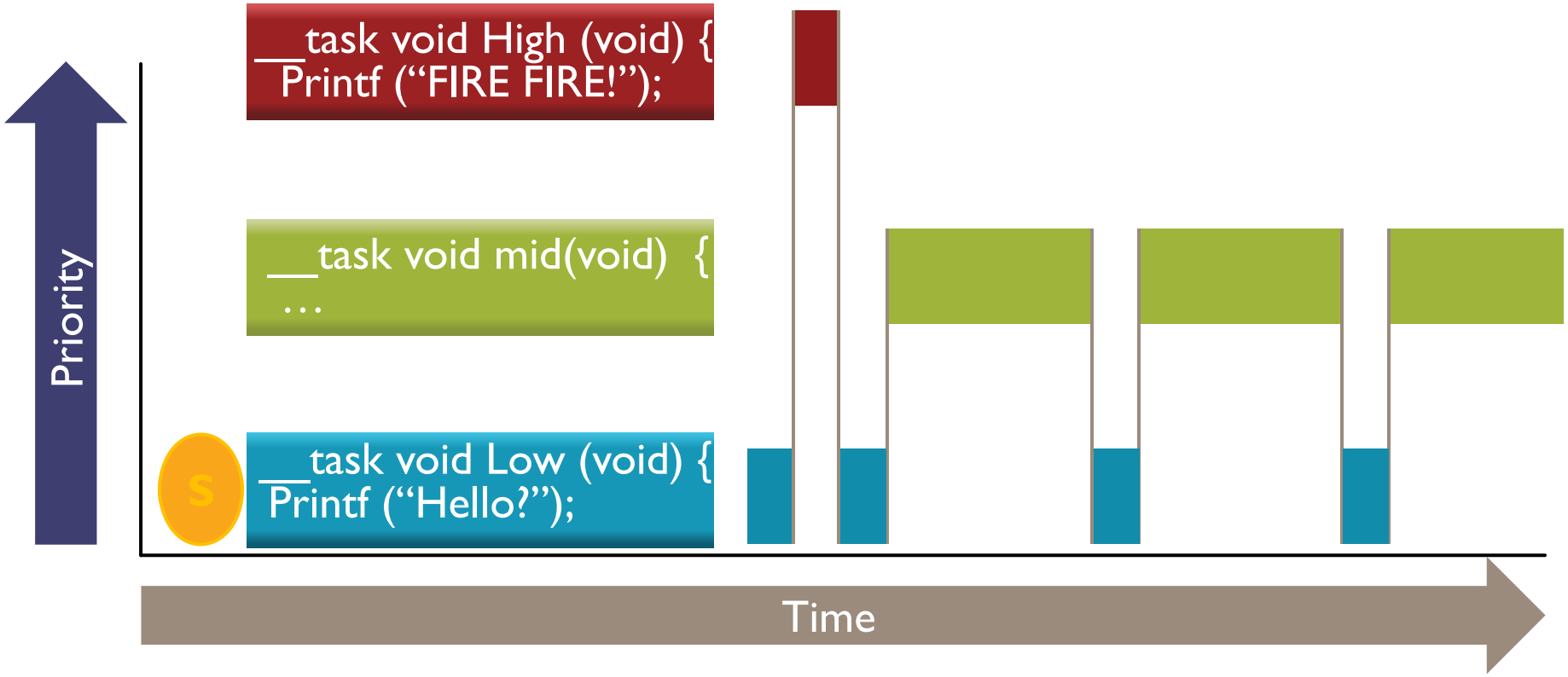
Preemption – Blocking and Semaphores

- Low() has semaphore, starts printing
- High() preempts low task, wants semaphore to print
  - Can't get semaphore because Low() can't finish printing



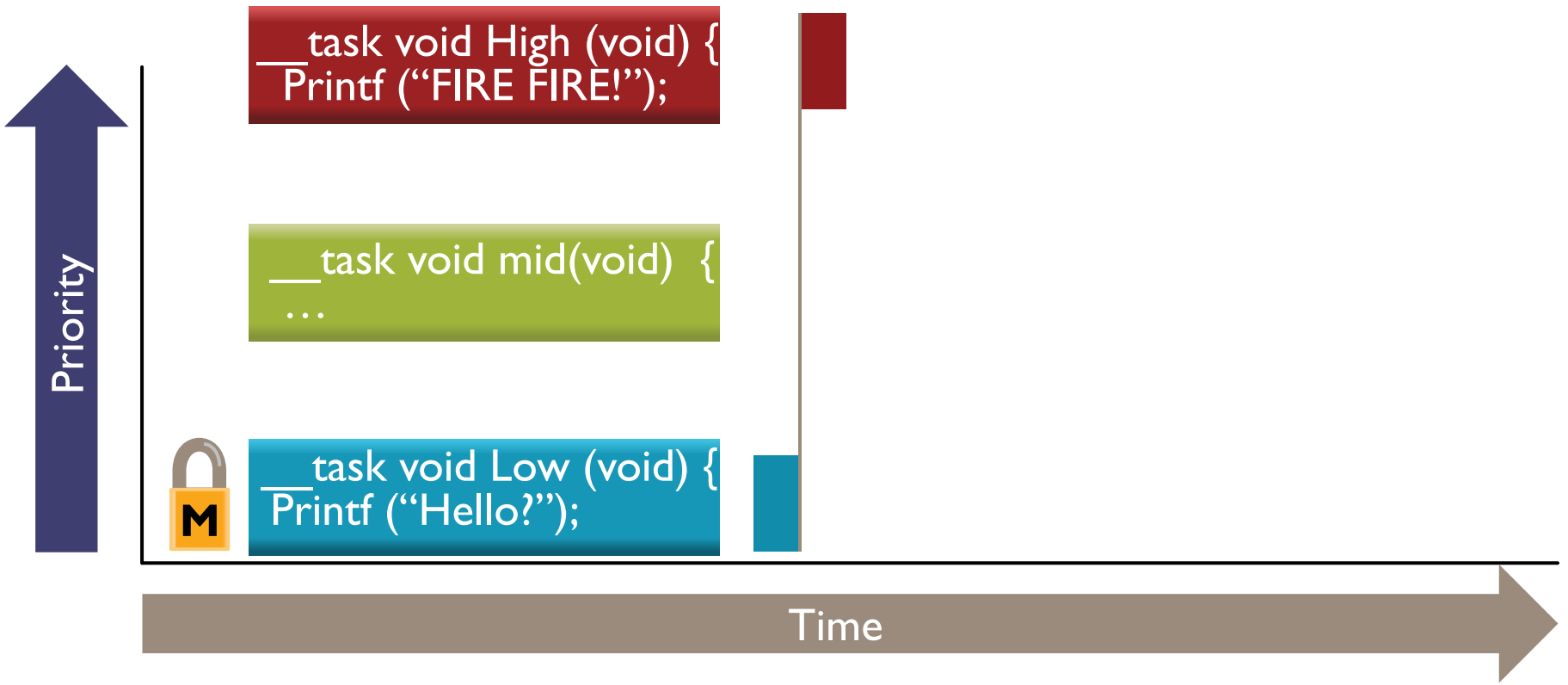
Preemption – Blocking and Semaphores

- Low() has semaphore, starts printing
- High() preempts low task, wants semaphore to print
  - Can't get semaphore because Low() can't finish printing



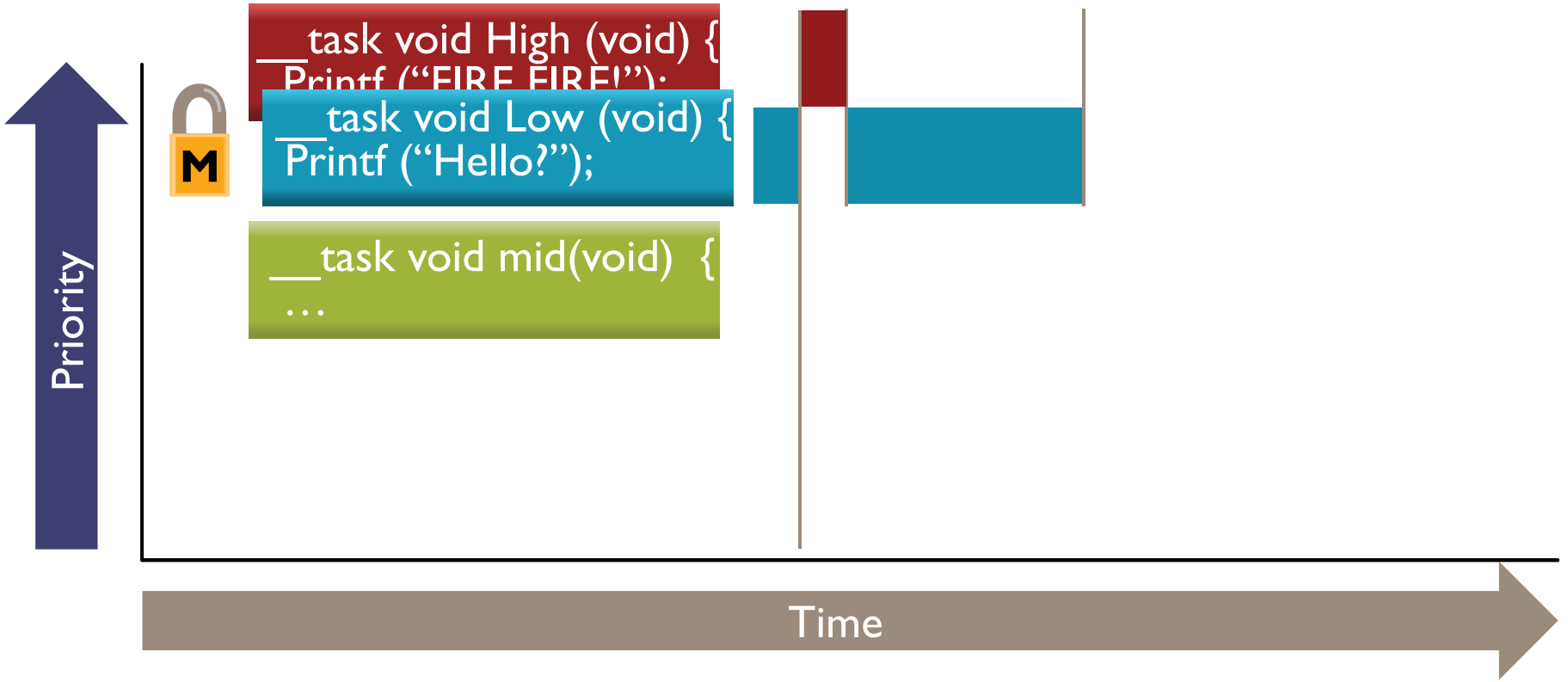
Preemption – Blocking and Mutexes

- Low() has locked the mutex, starts printing
- High() preempts low task, wants mutex to print
  - Can't get mutex because Low() can't finish printing



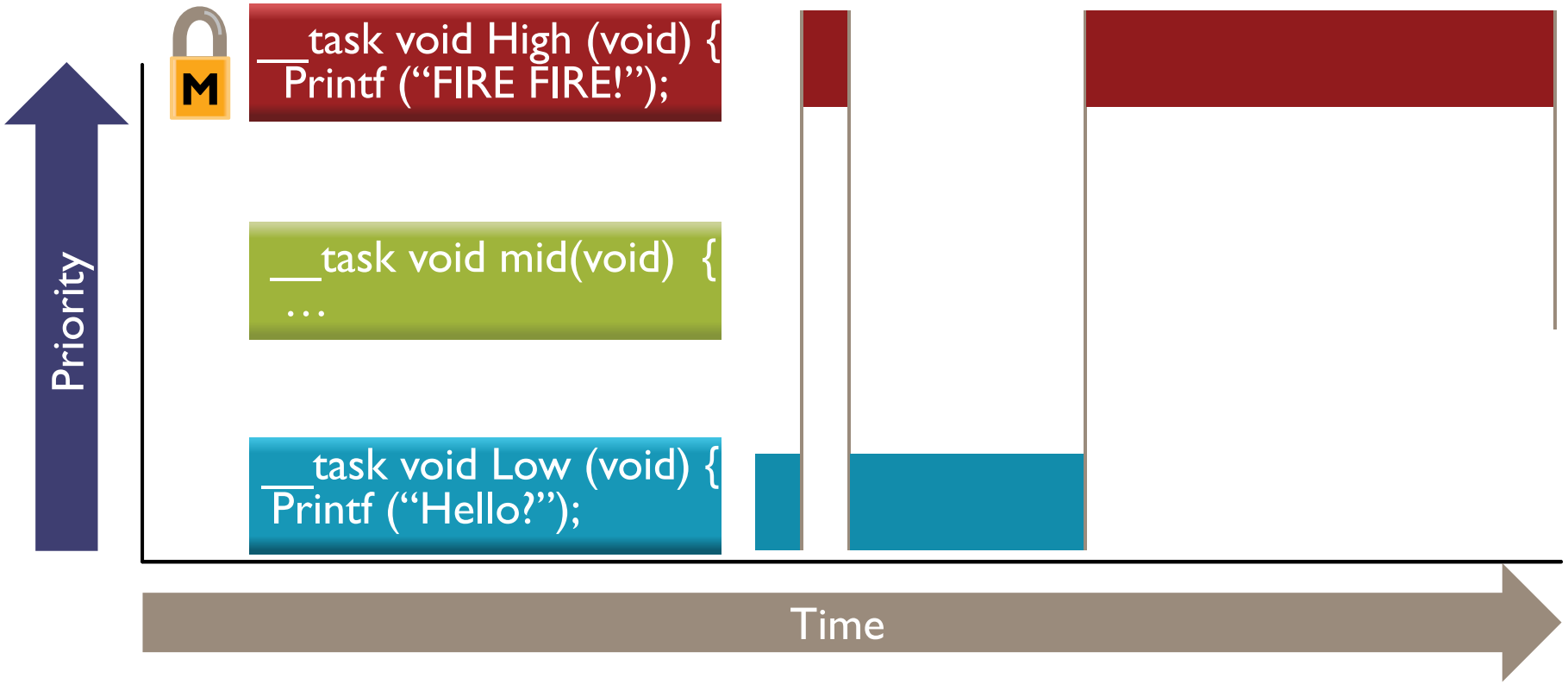
Preemption – Priority Inheritance

- The mutex raises the priority of the Low(),since it is lower than the priority of the calling task
- Low() finishes printing



Preemption – Priority Inheritance

- Low() unlocks the mutex
- Low() returns to normal priority
- High() locks the mutex and prints the message



THE END

That gives a brief introduction to ARM RTX !

Remember – it is FREE !

Really FREE….

ARM University program supports you and RTX !

Other Examples

Evaluation tools can be found at:

www.keil.com/demo

RTX examples can be found on your hard drive at:

C:\keil\ARM\RL\RTX\Examples\

RL-ARM Getting Started Guide:

http://www.keil.com/rl-arm/rl-gettingstarted.asp

Little Book of Semaphores by Allen B. Downey (Third party book):

greenteapress.com/semaphores

PRIMARY

18 140 171	149 186 205	200 217 227
159 180 59	203 210 150	227 230 199
145 27 29	190 127 108	218 182 169
154 139 124	197 187 178	222 217 211
55 55 104	125 128 158	180 180 199

ACCENT

18 140 171	253 201 134	254 225 186
159 180 59	234 157 182	246 203 218
145 27 29	109 102 95	169 161 155