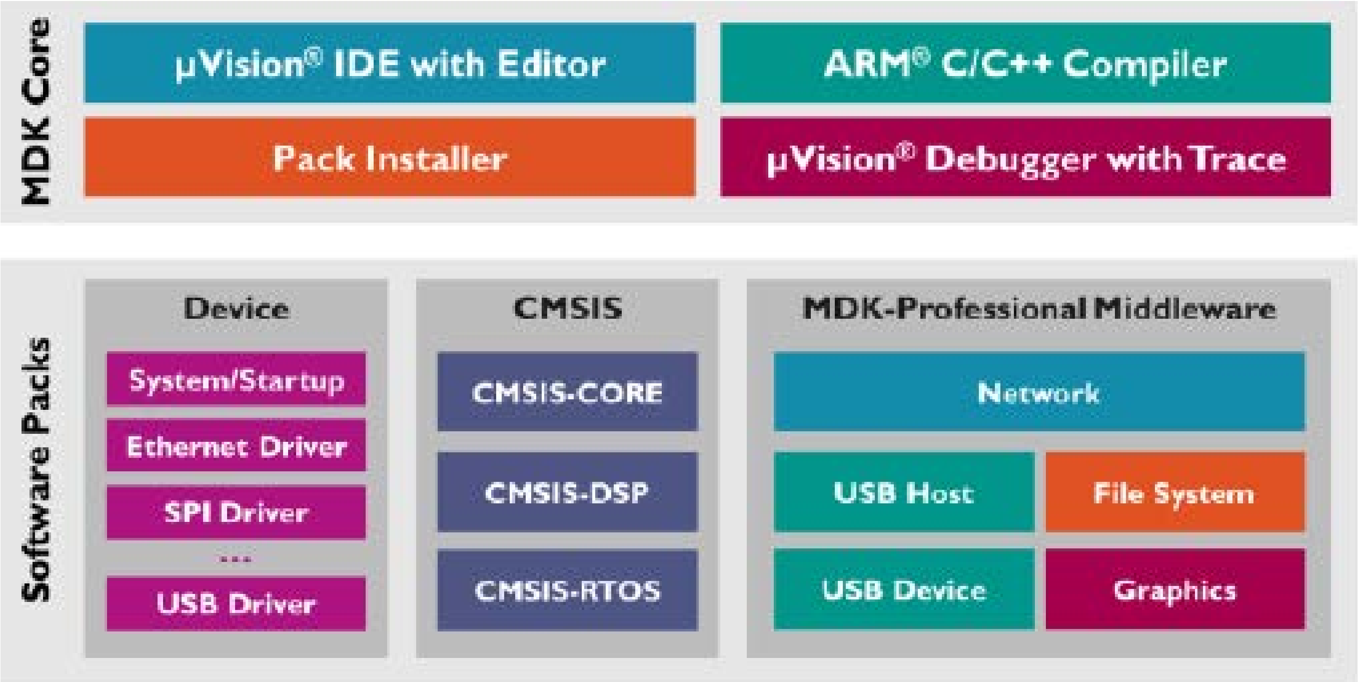
CMSIS Real Time Operating System (Based on Keil RTX)

References: HTTPS://developer.mbed.org/handbook/CMSIS-RTOS
http://www.keil.com/pack/doc/cmsis/rtx/html/index.html

In Keil directory: *C:/Keil/ARM/Pack/CMSIS/4.1.1/CMSIS-RTX/index.html*
(user code templates, examples, documentation)

CMSIS-RTOS



MDK Core

µVision® IDE with Editor

ARM® C/C++ Compiler

Pack Installer

µVision® Debugger with Trace

Software Packs

Device

System/Startup

Ethernet Driver

SPI Driver

...

USB Driver

CMSIS

CMSIS-CORE

CMSIS-DSP

CMSIS-RTOS

MDK-Professional Middleware

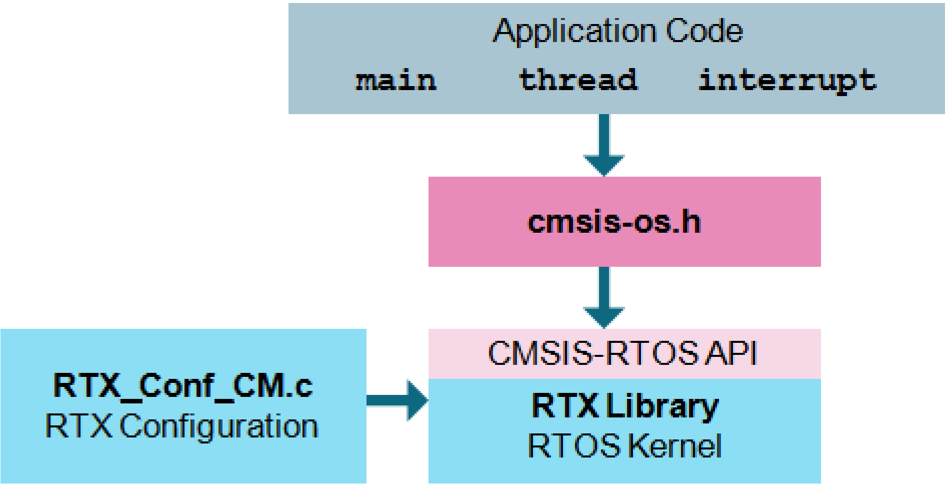
Network

USB Host

File System

USB Device

Graphics



Application Code

`main` `thread` `interrupt`

cmsis-os.h

CMSIS-RTOS API

RTX Library
RTOS Kernel

RTX_Conf_CM.c
RTX Configuration

CMSIS-RTOS RTX Structure

Using CMSIS-RTOS in a project

- Create the project
- Click *Manage Run-Time Environment* button
  - Or from menu: *Project > Manage > Run-Time Environment*
- Select: *CMSIS > RTOS (API) > Keil RTX*
  - Adds RTX configuration file to project: *RTX_Conf_CM.c*
  - Adds Cortex-M4 RTX library to project: *RTX_CM4.lib*
  - Adds API file to project: *cmsis_os.h*
  - In main program add: *#include <cmsis_os.h>*

- Edit RTX options in *RTX_Conf_CM.c* (next slide) to configure the RTX kernel for the particular application

RTX_Conf_CM.c – Kernel Configuration

Edit RTX parameters to tailor the kernel to the project

- *OS_TASKCNT* = # concurrent running threads (in any state)
- *OS_PRIVCNT* = # threads with user-provided stack
- *OS_STKSIZE* = stack size for each thread
- *OS_STKCHECK* = enable/disable status checking (of stack)
- *OS_SYSTICK* = 1 to use Cortex SysTick timer as RTX Kernel Timer
- *OS_CLOCK* = timer clock frequency [Hz]
- *OS_TICK* = timer tick interval [µs]
- *OS_ROBIN* = 1 enable round-robin thread switching
- *OS_ROBIN* = 0 disable round-robin & use timer/event scheduling
- *OS_ROBINTOUT* = time slice for round-robin task switching
- *OS_TIMERS* = # of user timers (from on-chip timers)
- *os_idle_demon (void)*{} = idle task system thread

RTX kernel functions

- RTX kernel runs and executes the "main" thread at startup

```
#include <cmsis_os.h>     // CMSIS RTOS header file
…
void main () {  //main thread run by the kernel at startup
…  create other threads
…  other program code
}
```

- Kernel can be stopped to permit initializations before continuing to schedule threads
  - osKernelInitiaze()   - stop kernel to allow initializations
  - osKernelStart() – start kernel, executing threads in list

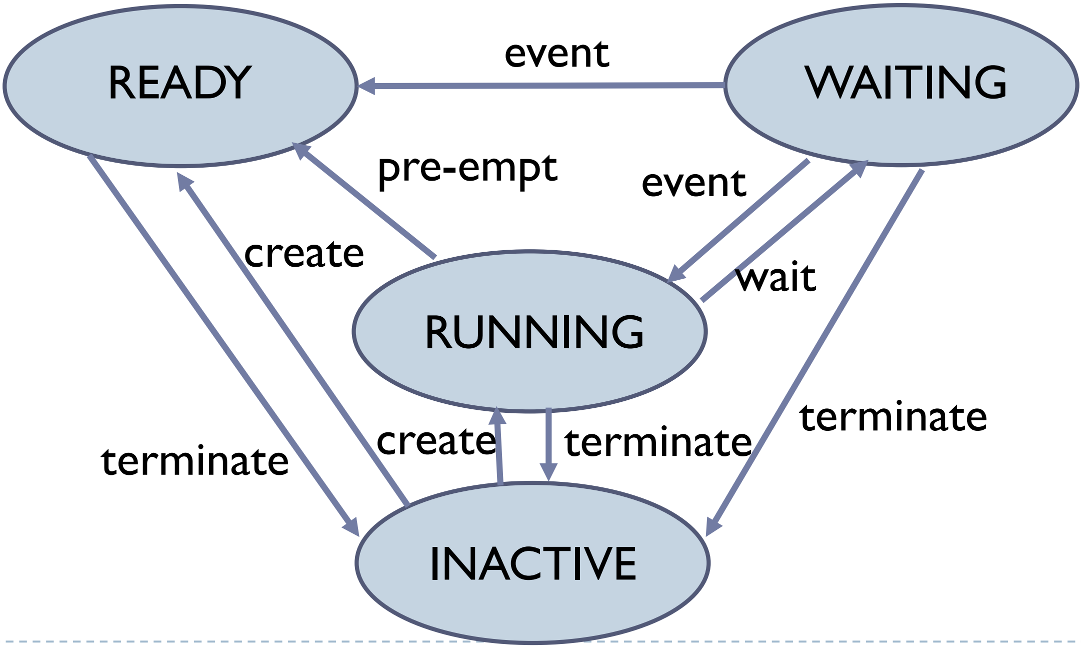
```
void main () {
   if (osKernelInitiaze() != osOK) {error routine}
   …  perform device/task/object initializations
   if (osKernelStart() != osOK) {error routine}
}
```

RTX Threads

- The scheduling unit is the "thread"
  - *osThreadId tid_thread1;* //thread ID of thread "thread1"
    - Scheduling, messages, events, etc. refer to a thread ID
    - Thread ID will be generated by *osThreadCreate()*
- Define thread objects with *osThreadDef* "macro":
  - #define osThreadDef(myfunc, priority, instances, stack_size);
    - This macro is placed outside of any function
    - Defines a thread object, but leaves it INACTIVE (not schedulable)
- Threads are dynamically created, started/stopped, etc.
  - *tid_thread1 = osThreadCreate(osThread(thread1), NULL);*
    - Create and put "thread1" into the Thread List
    - Macro *osThread(thread1)* returns the thread1 object structure
  - *osThreadTerminate()* – stop execution (make inactive)
  - *osThreadYield()* – pass execution to next thread in list (but still READY)
  - *osThreadGetId()* – return thread ID of current thread
- Main thread and osTimerThread are created automatically

Thread states

- RUNNING – thread currently running
- READY to run, RTX chooses highest-priority
- WAITING for some time/event
- INACTIVE – thread not started or deleted



```
/*----------------------------------------------------------------------------
 *      Thread 1 'Thread_Name': Sample thread
 *---------------------------------------------------------------------------*/

#include <cmsis_os.h>      // CMSIS RTOS header file

void Thread (void const *argument);                   // thread function prototype
osThreadId tid_Thread;                                // thread id variable
osThreadDef (Thread, osPriorityNormal, 1, 0);      // Macro: define thread object

int Init_Thread (void) {

  tid_Thread = osThreadCreate (osThread(Thread), NULL);  //create the thread
  if(!tid_Thread) return(-1);

  return(0);
}

/* Define the thread routine */
void Thread (void const *argument) {
  while (1) {
    ; // Insert thread code here...
    osThreadYield();    // suspend thread
  }
}
```

```
/* Simple program using round-robin multitasking with two threads */
#include "cmsis_os.h"    // CMSIS-RTOS header file

int counter1, counter2;
osThreadId tid_job1;
osThreadId tid_job2;

void job1 (void const *arg) {    //First thread
    while (1) { // loop forever
        counter1++; // update the counter
    }
}
void job2 (void const *arg) {    //Second thread
   while (1) { // loop forever
       counter2++; // update the counter
    }
}

osThreadDef (job1, osPriorityAboveNormal, 1, 0);   //thread object "job1"
osThreadDef (job2, osPriorityAboveNormal, 1, 0);   //thread object "job2"

int main (void) {
    osKernelInitialize ();                                        // setup kernel (suspend kernel for now)
    tid_job1 = osThreadCreate (osThread(job1), NULL);    // create and add thread "job1" to Thread List
    tid_job2 = osThreadCreate (osThread(job2), NULL);    // create and add thread "job2" to Thread List
    osKernelStart ();                                             // start kernel
}
```

Thread priorities

- Priority levels

osPriorityIdle	(-3) – lowest priority
osPriorityLow	(-2)
osPriorityBelowNormal	(-1)
osPriorityNormal	(0) – default priority
osPriorityAboveNormal	(+1)
osPriorityHigh	(+2)
osPriorityRealTime	(+3) – highest priority

- Thread priority set when thread object defined:
  - #define osThreadDef (function, priority, #threads, stack size);
  - Main thread given priority osPriorityNormal
- Change priorities:
  - osThreadSetPriority(tid, p); //tid = task id, new priority p
  - osThreadGetPriority(); //return current task priority

ARM CMSIS-RTOS scheduling policies

- Round robin schedule (OS_ROBIN = 1)
  - All threads assigned same priority
  - Threads allocated a fixed time
    - OS_SYSTICK = 1 to enable use of the SysTick timer
    - OS_CLOCK = CPU clock frequency (in Hz)
    - OS_TICK = "tick time" = #microseconds between SysTick interrupts
    - OS_ROBINTOUT = ticks allocated to each thread
  - Thread runs for designated time, or until blocked/yield
- Round robin with preemption (OS_ROBIN = 1)
  - Threads assigned different priorities
  - Higher-priority thread becoming ready preempts (stops) a lower-priority running thread
- Pre-emptive (OS_ROBIN = 0)
  - Threads assigned different priorities
  - Thread runs until blocked, or executes osThreadYield(), or higher-priority thread becomes ready (no time limit)
- Co-operative Multi-Tasking (OS_ROBIN = 0)
  - All threads assigned same priority
  - Thread runs until blocked (no time limit) or executes osThreadYield();
  - Next ready thread executes

Preemptive multitasking

- *#define OS_ROBIN 0*   in *RTX_Conf_CM.c*
- RTX suspends the running thread if a higher priority thread (HPT) becomes ready to run
- Thread scheduler executes at system tick timer interrupt.
- Thread context switch occurs when:
  - Event set for a HPT by the running thread or by an interrupt service routine (event for which the HPT was waiting)
  - Token returned to a semaphore for which HPT is waiting
  - Mutex released for which HPT is waiting
  - Message posted to a message queue for which HPT is waiting
  - Message removed from a full message queue, with HPT waiting to send another message to that queue
  - Priority of the current thread reduced and a HPT is ready to run

Round-Robin Multitasking

- RTX gives a time slice to each thread (OS_ROBINTOUT)
- Thread executes for duration of time slice, unless it voluntarily stops (via a system "wait" function)
- RTX changes to next ready thread with same priority
  - if none – resume current thread

- Configure in *RTX_Conf_CM.c*
  - #define OS_ROBIN 1
  - #define OS_ROBINTOUT n

    n = #timer ticks given to each thread/ "timeout" value

Basic wait/delay function

- Suspend a thread for a designated amount of time
- *osStatus osDelay (T);*
  - Change thread state to WAITING
  - Change thread state back to READY after T milliseconds
  - Return status = *osEventTimeout* if delay properly executed
    = *osErrorISR* if osDelay() called from an ISR (not permitted)

```
#include "cmsis_os.h“

void Thread_1 (void const *arg) { // Thread function
    osStatus status;          // capture the return status
    uint32_t delayTime;   // delay time in milliseconds
    delayTime = 1000;     // delay 1 second
     :
     status = osDelay (delayTime); // suspend thread execution
}
```

Inter-thread communication

- Signal flags – for thread synchronization
  - Each thread can have up to 31 SFs.
  - A thread can wait for its SFs to be set by threads/interrupts.
- Sempahores – control access to common resource
  - Semaphore object contains tokens ("counting" semaphore)
  - Thread can request a token (put to sleep if none available)
- Mutexes – mutual exclusion locks
  - "lock" a resource to use it, and unlock it when done
  - Kernel suspends threads that need the resource until unlocked
- Message Queues and Mail Queues
  - Queue is a first-in/first-out (FIFO) structure
  - "Message" is an integer or a pointer to a message frame
  - "Mail" is a memory block to put on queue/get from queue
  - Suspend thread if "put" to full queue or "get" from empty queue

Signal Flags

- Signal flags not "created" – a 32-bit word of signal flags exists automatically within each thread.
- Signals are sent to a **thread** (using its thread ID)
- osSignalSet(tid, flags) – set SFs of thread tid
- osSignalClear(tid, flags) – clear SFs of thread tid
  - flags = int32_t; each "1" bit in "flags" sets/clears the corresponding SF
  - Example: flags=0x8002 => set/clear SF #15 and SF #0
  - Return int32_t, containing **previous** flags of tid
- osSignalWait(flags, timeout)
  - Wait for SFs corresponding to "1" bits in "flags" to be set, or until timeout
    - timeout = 0 if no wait time desired
  - Return osEventSignal if designated SFs are set
    osEventTimeout if no signal before timeout
    osOK if timeout=0 and no signal

Mutual Exclusion (MUTEX)

Provide exclusive access to a resource

- osMutexDef(m1); //Macro: MUTEX object definition
- *osMutexId m_id;* //MUTEX ID
- *m_id = osMutexCreate(osMutex (m1));* //create MUTEX obj
- *status = osMutexDelete(m_id);* //delete MUTEX obj
- *status = osMutexWait(m_id, timeout);*
  - Wait until MUTEX available or until time = "timeout"
    - timeout = 0 to return immediately
    - timeout = osWaitForever for infinite wait
  - "status" = osOK if MUTEX acquired

    osErrorTimeoutResource if not acquired within timeout

    osErrorResource if not acquired when timeout=0 specified
- *status = osMutexRelease(m_id);* //release the MUTEX

Timeout arguments for other objects have same options

Semaphores

Allow up to t threads to access a resource

- *osSemaphoreDef(s1)* //Macro: define semaphore object s1
- osSemaphoreId s_id; //semaphore ID
- s_id = osSemaphoreCreate(osSemaphore(s1, t));
  - Create s1 and set initial #tokens = t
- osSemaphoreDelete(s_id); //delete the semaphore
- ntok = osSemaphoreWait(s_id,timeout);
  - Wait until token available, or until timeout
  - Return #tokens that were available
  - If ntok >0, token is obtained and #tokens is decremented
  - If #tokens=0, no token was available at timeout
- osSemaphoreRelease(s_id); //increment #tokens in s1

Message queues

“Message” = a 32-bit integer or a 32-bit pointer

- *osMessageQId q_id;* // *ID of queue object*
- *osMessageQDef (name, queue_size, type);* //Macro: define message queue object
  - queue_size = max #messages in the queue
  - type = 32-bit data type of message (32-bit integer or pointer)
- *q_id = osMessageCreate(osMessageQ(name), NULL);*
  - Create and initialize a message queue, return queue ID
- *status = osMessagePut(q_id, msg, timeout);*
  - Add “msg” to queue; wait for “timeout” if queue full
  - Status = *osOK* : msg was put into the queue

    = *osErrorResource* : no queue memory available

    = *osErrorTimeoutResource* : no memory available at timeout
- *status = osMessageGet(q_id, timeout);*
  - Get message from queue; wait for “timeout” if no message
  - Status = *osOK* : no msg available and timeout=0

    = *osEventTimeout* : no message available before timeout

    = *osEventMessage* : msg received *(“status” is a “union” structure)*

    pointer = status.value.p

    value = status.value.v

Mail queues

Send/receive messages other than single integer/pointer

- *osMailQId q_id;* // *ID of queue object*
- *osMailQDef (name, queue_size, type);* // Macro: define mail queue object, size, type
- *q_id = osMailCreate(osMailQ(name), NULL);*
  - Create and initialize a message queue, return queue ID
- *mptr = osMailAlloc(q_id, timeout);* *(osMailCAlloc() – allocate and clear memory)*
  - *Allocate a memory block in the queue that can be filled with mail info*
  - *"mptr" = pointer to the memory block (NULL if no memory can be obtained)*
  - *Wait, with timeout, if necessary for a mail slot to become available*
- *status = osMailFree(q_id, mptr); - free allocated memory*
- *status = osMailPut(q_id, mptr);*
  - Add mail (pointed to by mptr) to queue; wait for "timeout" if queue full
  - Status = *osOK* : mail was put into the queue
    = *osErrorValue* : mail was not allocated as a memory slot
- *status = osMailGet(q_id, timeout);*
  - Get mail from queue; wait for "timeout" if no mail available
  - Status = *osOK* : no mail available and timeout=0
    = *osEventTimeout* : no mail available before timeout
    = *osEventMail* : mail received, pointer = value.p

CMSIS-RTOS RTX examples

- Examples throughout the CMSIS-RTOS reference
  - C:/Keil/ARM/Pack/ARM/CMSIS/version#/CMSIS_RTX/Doc/index.html
  - Examples are included in the CMSIS-RTOS API reference description of each function
- Code templates (C files) for most features are provided in
  - C:/Keil/ARM/Pack/ARM/CMSIS/version#/CMSIS_RTX/UserCodeTemplates
- RTX Blinky example in:
  - C:/Keil/ARM/Pack/Keil/STM32F4xx_DFP\1.0.6\Boards\ST\STM32F4-Discovery