SPI,
DMA,
& Real-time Operating Systems

February 18, 2013

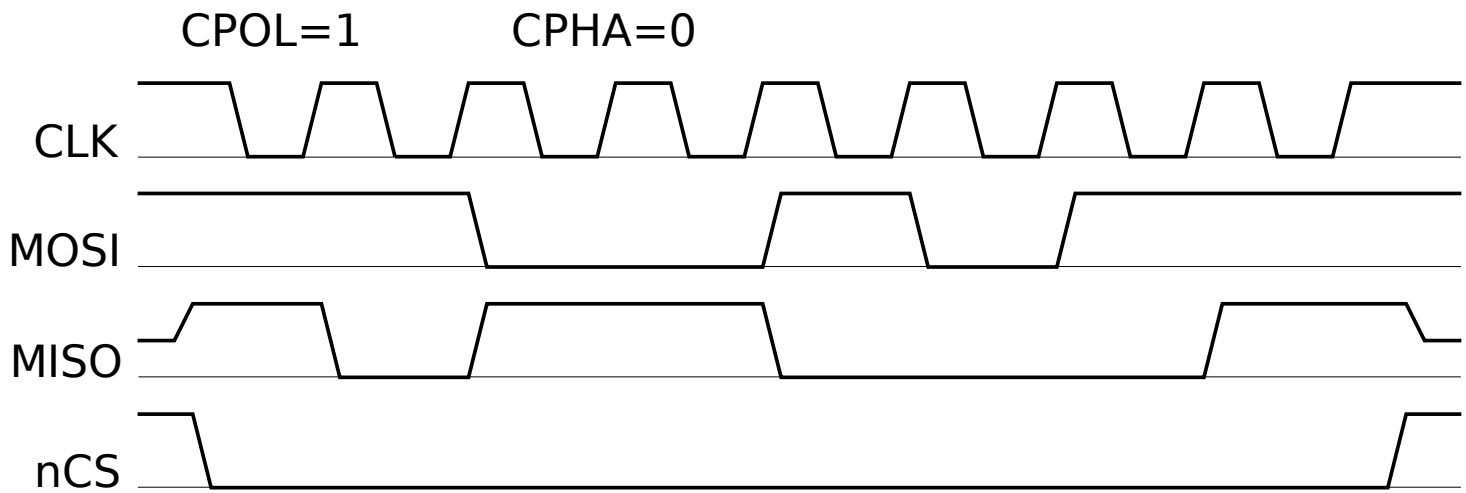
Table of contents

Serial Peripheral Interface (SPI)

- Common, 4-pin, multi-slave, full-duplex bus
- Capable of high speeds and uses a very simple hardware interface
- Master generates clock
- Both master and slave have a data output line to provide data on the clock
- Slaves selected by active-low select line (nCS, nSS) – one per slave
- Data always goes in both directions – send *dummy bytes* if nothing to send

SPI Parameters

- Clock speed – usually 1-30MHz
- Clock polarity (CPOL) – value of clock line when idle
- Clock phase (CPHA) – which edge (first or second) latches new data
  - If CPHA $= 1$, first edge reads data, second edge shifts new data to transmit



SPI Virtual-Register Protocol

- Most SPI slaves implement a virtual-register based protocol
- First byte contains address, a read/write flag, and a burst flag
- Subsequent bytes contain data to be transferred
- Burst indicates that multiple bytes will be accessed sequentially and the address should be automatically incremented

SPI Virtual-Register Protocol

Read Example

Master	Addr & R/nW & Burst			
Slave		Data1	Data2	Data3

Write Example

Master	Addr & Burst	Data1	Data2	Data3
Slave				

Direct Memory Access (DMA)

- AHB matrix in STM32F4 has both master and slave devices
- Slave – Cannot initiate transactions
  - RAM
  - Flash
  - Peripherals
- Master
  - CPU (multiple masters)
  - USB
  - Ethernet
  - *DMA1 & DMA2* (2 masters each)

DMA Motivations

- For use of serial peripherals ($<$1MHz word rate), it is a waste of CPU time to move data
- Wait thousands of cycles for byte to be complete $\rightarrow$ copy byte $\rightarrow$ repeat
- $\uparrow$ dumb
- DMA is a dumb device dedicated to this task
- It knows when and where to move data and how to raise an interrupt for the CPU when a specified amount of data has been moved
- As a bus master, it has the power to perform all the same transactions that the CPU can (including memory-to-memory transfers)

Alternate DMA Motivations

- Some devices such as USB, Ethernet, and audio may require data to be moved as fast as the CPU possibly could
- This would leave the CPU saturated and with no time to run your application
- Memory-to-memory copies can also be done efficiently with DMA if configuration time is tolerable

DMA Structure

Streams

- A stream contains all of the resources to move data from one place to another
- Up to 8 may be active at once per DMA device – Use priority-based arbitration
- Each stream gets its own interrupt vector

Channels

- Essentially a trigger source, associating a stream with a peripheral
- Allow the peripheral to indicate when it is ready for another transfer

DMA Structure

FIFOs

- 4-word buffer for packing/unpacking to optimize memory bandwidth
  - If AHB nearly saturated, preferable to move one 32-bit value rather than 4 8-bit values
- Bus transfers initiated at 1/4, 1/2, 3/4 or full
  - Burst transfers can be faster as arbitration may be held across several bytes

DMA Configuration

- Configuration is just like any other peripheral in the STM32 ecosystem
- Parameters:
  - Peripheral address – The address of the memory-mapped peripheral register to read or write
  - Memory address – The base address for the memory transfer
  - Direction
  - Request channel (Table 20/21 in FRM)
  - Number of words
  - Word size
  - Circular/double buffering
  - Priority
  - FIFO configuration (or direct mode)
  - Interrupt configuration

DMA Buffering Options

Circular Buffering

- DMA will continue when it finishes, resetting the memory address and count each time
- Use when maximum throughput is required (no overhead restarting transfer)
- Interrupts can be generated at different points in the transfer to allow time to reload/read buffer

Double Buffering (Implies Circular)

- Two base addresses are provided and DMA switches between the two
- If the two buffers are adjacent, this is the same as circular buffering

Memory-to-peripheral Transfers

- Peripheral must indicate to DMA controller (request) when new data required
- `SPI_DMACmd(SPI1, SPI_DMAReq_Tx, ENABLE);`
- Generally want memory pointer to increment, peripheral pointer constant
- Unless the target peripheral has a FIFO built in, must transfer one word at a time
- Set word size to be data size for peripheral

Peripheral-to-memory Transfers

- Peripheral must indicate to DMA controller (request) when new data available
- `SPI_DMACmd(SPI1, SPI_DMAReq_Rx, ENABLE);`
- Generally want memory pointer to increment, peripheral pointer constant
- May buffer multiple words in DMA FIFO
- Set word size to be data size for peripheral

Full-Duplex with DMA

- Requires two streams
- For SPI master, read-through-write creates dependency of RX on TX
  - Have to start RX DMA before TX

Real-time Operating Systems (RTOS)

An RTOS....

- ... provides basic multi-tasking support (threads)
- ... provides resource management
- ... provides thread synchronization utilities
- ... provides varying degrees of hardware abstraction

You will be using Keil's RTX with a CMSIS-RTOS API wrapper

REFERENCE MATERIAL

Anything I don't cover about this:

`http://bennahill.com/docs/cmsis/CMSIS/Documentation/RTOS/html/`

Threads

- Thread support allows multiple tasks to run concurrently
- A *scheduler* determines which should be running and for how long
  - For this course, scheduler is *round-robin*
- Threads have their own stack which allows them persistent data storage

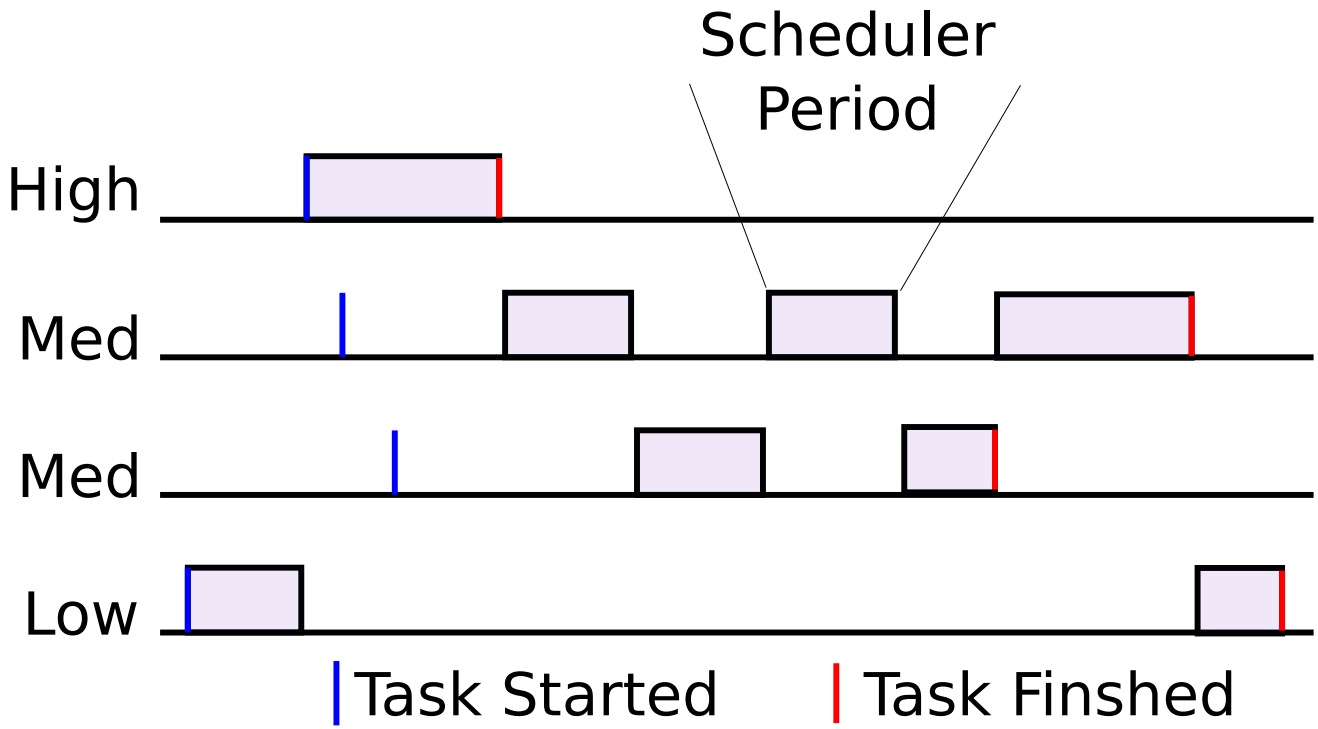
Round-Robin (RR) Scheduler

- Simple *priority-based*, *time-sharing*, *pre-emptive* scheduling scheme
- Very common among low-power embedded systems
- *Period* or *tick rate* refers to the frequency of context switches
  - Faster means more even distribution among tasks but more overhead in context switches

Schedule

- Tasks ready to execute in *ready pool*
- Highest-priority threads in ready pool are cyclically given a fixed time to execute
- Threads sleep while delaying or waiting on a resource or synchronization

RR Scheduling Example



Priority Selection

- Objective is to have *all* tasks *always* finish when they need to
- For this class, use a fixed-priority scheme
- Interrupts are at a higher priority than any thread
  - Limit time here to minimize delays in threads
- All non-real-time tasks (no deadline) minimum priority
- For others, prioritize based on how quickly it must finish

Soft Timers

Differ from "hard" timers in that they don't use a hardware timer directly and are instead managed by scheduler ticks

- Don't necessarily have guarantees regarding the actual time of a delay
- Generally limited to the granularity of the scheduler tick
- Upon expiration, a callback function is executed, usually at a high priority (like an interrupt)

One-time A single timer that expires only once and stops

Periodic Reloads value upon expiration and maintains periodic behavior

CMSIS-RTOS Soft-Timer API

```
typedef struct {
    os_ptimer ptimer; // Callback function address
                      // (takes void * parameter)
} osTimerDef_t;

typedef enum {
    osTimerOnce = 0,
    osTimerPeriodic = 1
} os_timer_type;

osTimerId osTimerCreate(osTimerDef_t *timer_def,
                        os_timer_type type, void *argument);

osStatus osTimerStart (osTimerId timer_id, uint32_t millisec);

osStatus osTimerStop (osTimerId timer_id);
```

Thread Management

`http://bennahill.com/docs/cmsis/CMSIS/Documentation/RTOS/html/`

Creation

- Macro shortcut: `osThreadDef (name,priority,instances,stacksz);`
  - Make a thread with specified priority and create a few instances with statically allocated stacks
-

```
osThreadId osThreadCreate(osThreadDef_t *thread_def, void *argument)
```

Basic Operations

Yield	Give up the alloted scheduler slot for now – will get it back
Terminate(id)	Kill a thread
SetPriority(p)	Assign a new priority

Resource Management

"Resources"

Generally, items that you need to control access to because multiple threads might want to use simultaneously

Could be:

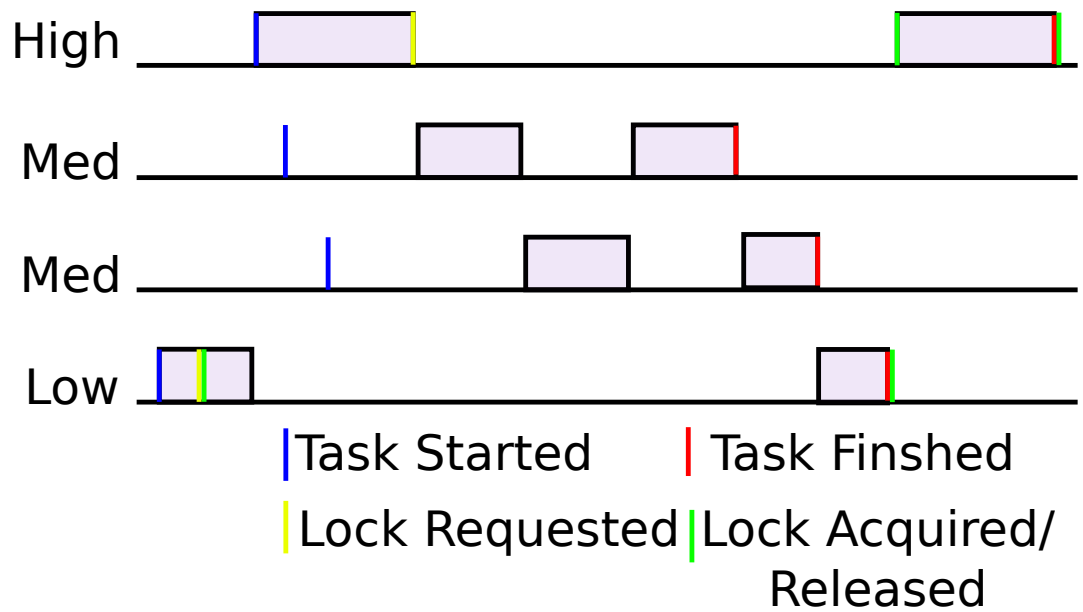
- Variables or memory regions (i.e. sensitive data structures like linked lists)
- Peripherals (don't want to interrupt transmissions)
- Flow control and synchronization resources

Mutual Exclusion (Mutex)

- At most one thread may hold a mutual exclusion "lock"
- Other threads that wait will sleep until lock is released
- When a lock is released, it is given to the highest priority waiting thread
- `osMutexId osMutexCreate (osMutexDef_t *mutex_def);`
- `osStatus osMutexWait (osMutexId mutex_id, uint32_t millisec);`
- `osStatus osMutexRelease (osMutexId mutex_id);`

Priority Inversion

A basic fixed-priority scheduler has potential problems with priority inversion, where a pair of tasks run with inverted relative priorities



Note that the medium priority tasks, even without holding any shared resources, prevent the high-priority task from executing

Priority Inheritance Protocol

To fix this, most RTOS implementations use Priority Inheritance Protocol (PIP)

- When a task holds a mutex, it runs with the priority of the highest-priority waiting thread
- This effectively prioritizes the release of the lock according to those that need it
- This isn't perfect since problems can come up in nested cases, but it's easy and requires minimal prior knowledge

Semaphores

Similar to a mutex, but does not associate ownership by any thread

```
osSemaphoreId osSemaphoreCreate (osSemaphoreDef_t *semaphore_def,
int32_t count);
```

- Associates well with countable resources
- Two basic operations:
  - Wait – Blocks as long as the value of the semaphore is 0, then decrement

    ```
    int32_t osSemaphoreWait(osSemaphoreId semaphore_id,
    uint32_t millisec);
    ```
  - Release – Increment the value of the semaphore

    ```
    osStatus osSemaphoreRelease(osSemaphoreId semaphore_id);
    ```

Semaphores with a maximum count of 1 are called *binary semaphores*

Mutex vs. Semaphore

- Semaphores are faster in every way – very quick locks (protecting an operation that may only take a few cycles) can take advantage of this
- Semaphores have no knowledge of thread ownership and thus can't use PIP
- Semaphores can easily associate with multiple identical functional units

Thread Synchronization

Most RTOSs provide a number of ways to safely and efficiently move data of various sizes between threads

- The names and conventions of each of these devices varies greatly across different OS implementations
- Generally: Queues (arbitrarily-sized data), events (minimal data), and something in between (usually integers/pointers)

Events (CMSIS-RTOS "Signals")

These are minimally sized data intended purely to indicate that something has happened

- Used for timing and very simple I/O
- Implemented as a 32-bit integer where each bit represents a different application-designated signal

Basic Operations

set(thread, flag mask) Set a bit mask on a thread – OR its current signal flags with your provided mask

wait(flag mask) Wait for the specified signal flags to be set – Clear selected flags once all have been set.

Queues (CMSIS-RTOS "Mail Queue")

Send copies of arbitrarily sized data to a mailbox

- Includes efficient (fixed block size) memory allocation pool
- Often used for high-level gatekeeping tasks (i.e. dispatching or receiving large packets of data from a peripheral, perhaps accelerometer data...)
- Queue length limited by number of buffers

Basic Operations

allocate	Allocate a block from the pool and return a pointer to it so it can be filled with data – this must later be freed
put(data)	Add a block to the queue
get	Blocks until data available – Pointer to data returned (you should free it now)
free(data)	Free a block now that it has been received

Message Queue

A simpler queue which just sends integers

- Flexible for you to design your own more efficient protocol
- No explicit allocation required but messages can be pointers
- Queue length limited by a buffer size

Basic Operations

put (uint) Add data to the queue

get Wait for data in queue and return it

Hardware Abstraction

CMSIS-RTOS provides no real hardware abstraction