ECE3849
D-Term 2021

Real Time Embedded Systems

Module 2 Part 3

Module 2 Part 3 Overview

- Handling Shared Data
  - Atomic Operations
    - Avoid multiple read / write
    - Cortex-M4 Bit Banding functions
  - Inter-thread communications
    - Binary Semaphore
    - Mailboxes
    - Buffering

Atomic Operations

- Operations that can be completed in a single instruction are called atomic operations.
- Global variables that are larger than a single 32-bit CPU register are not atomic in the Cortex-M4.
- For other processors, this may be different.
  - You need to know the number of bits in your primitive data type.
  - You need to know CPU architecture.
    - How many bits is the CPU?
    - Does it have special instructions that might allow normally non-atomic operations to be atomic?
  - Boolean and char are always reads and writes are always atomic independent of hardware architecture.
- Using type definitions helps
  - Using type definitions like uint32_t makes code clearer and more portable.
  - Using int, short, long which may very in length between 16, 32, and 64-bit processors.

Atomic Operations

- Are these Atomic or Non-Atomic?
  - x++
    - ?
  - x += y;
    - ?
  - x = y;
    - ?
  - if (x > y) { x = 0; }
    - ?
  - x |= 1 << 3;
    - ?
- Not sure check the disassembled code.

Read-Modify-Writes

- Watch out for implicit read-modify-writes
  - x += y; ➔ x = x + y; (reads x, adds y , writes x)
  - x--; ➔ x = x + 1; (reads x, adds 1, writes x)
  - Single line of code but multiple instructions.
  - If an interrupt occurs after the modify but before the write. The ISR gets over written and is lost.
- Some read-modify-writes may not have shared data problems.
  - main() (low priority task) is doing a read-modify-write.
  - ISR can read it once using an atomic read and store it in a local variable for further use in the ISR.

Cortex-M4F Bit-banding

- Bit-banding allows atomic writes to a single bit in a 32-bit integer.
- In Cortex-M4F, the HWREGBITW() MACRO performs bit-banding.
  - Non-Atomic: x |= 1 << 3;
    - x = x | 0x8; //0x8 = binary 1000
  - Atomic equivalent:
    - uint32_t x;
    - HWREGBITW(&x, 3) = 1; // sets third bit location
    - Not portable, specific to this hardware
      - To use: #include “inc/hw_types.h”

Read and Write Recommendations

- Never write to the same global twice in a row.
  - If a ISR reads between the writes, it may grab an intermediate value.
- Avoid reading the same global variable more than once if another thread writes it.

```
y = gValue + 5;
…
z = gValue + 8;
```

Interrupt occurs writes new value

z will not equal y + 3 as expected.

- If interrupts are infrequent, it may be possible to perform a non-atomic read.
  - This is accomplished by reading the desired value repeatedly until the same value is returned twice.
  - If the same value is read twice, you have verified an ISR did not occur during the operation.

Repeated Non-Atomic Reading

- Make the variable volatile to let the compiler know it might be changed at any time.

```
volatile int iHours, iMinutes, iSeconds; // shared variables

void TimerISR(void) {
    // clear interrupt flag
    if (++iSeconds >= 60) {
        iSeconds = 0;
        if (++iMinutes >= 60) {
            iMinutes = 0;
            if (++iHours >= 24)
                iHours = 0;
        }
    }
}

long SecondsSinceMidnight(void) { // called from low-priority thread
    int iHr, iMin, iSec;    // local copies of the shared data
    int iHr2, iMin2, iSec2; // second copy

    iHr = iHours; iMin = iMinutes; iSec = iSeconds; // read globals
    do {
        iHr2 = iHr; iMin2 = iMin; iSec2 = iSec;
        iHr = iHours; iMin = iMinutes; iSec = iSeconds;
    } while (iHr2 != iHr || iMin2 != iMin || iSec2 != iSec);

    return ((iHr * 60) + iMin) * 60 + iSec; // use local copies
}
```

TimerISR is only called once per second. (very slow rate)

Declare two sets of variables to hold the values from two sets of read operations.

If ISR occurred during these statements an incorrect value would be returned.

- Read them until two consecutive reads match.
- This confirms no ISR happened.

Calculates return value with confirmed good data.

Repeated Non-Atomic Reading

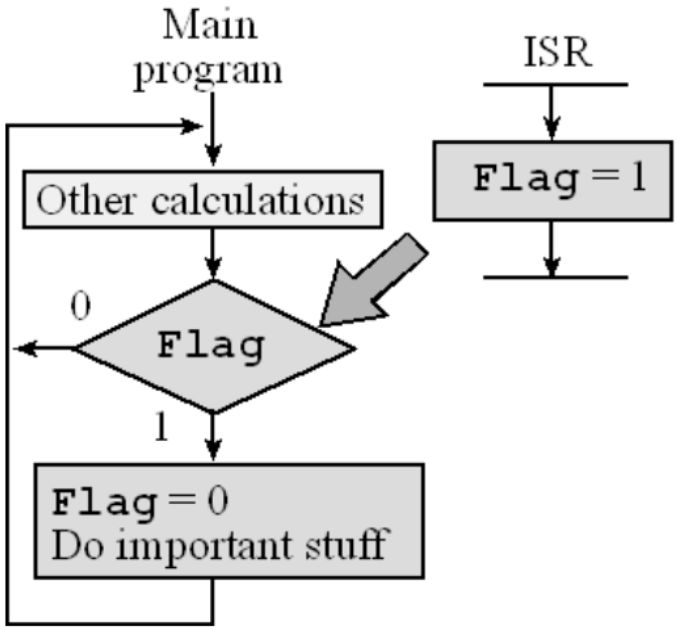
- What would happen if we left volatile out?
  - ?
- Why don't we get stuck in the loop?
  - ?

```
volatile int iHours, iMinutes, iSeconds; // shared variables

void TimerISR(void) {
    // clear interrupt flag
    if (++iSeconds >= 60) {
        iSeconds = 0;
        if (++iMinutes >= 60) {
            iMinutes = 0;
            if (++iHours >= 24)
                iHours = 0;
        }
    }
}

long SecondsSinceMidnight(void) { // called from low-priority thread
    int iHr, iMin, iSec;    // local copies of the shared data
    int iHr2, iMin2, iSec2; // second copy

    iHr = iHours; iMin = iMinutes; iSec = iSeconds; // read globals
    do {
        iHr2 = iHr; iMin2 = iMin; iSec2 = iSec;
        iHr = iHours; iMin = iMinutes; iSec = iSeconds;
    } while (iHr2 != iHr || iMin2 != iMin || iSec2 != iSec);

    return ((iHr * 60) + iMin) * 60 + iSec; // use local copies
}
```

- For complex data structures, this can be inefficient and error prone.
  - A sequence number can be added to the data.
  - In the ISR
    - The sequence number is incremented each time it is called
  - In the function
    - The sequence number is read.
    - The data is read.
    - The sequence number is read a second time and compared to the original.
    - If the sequence number is different, the data is re-read until the sequence number remains the same.

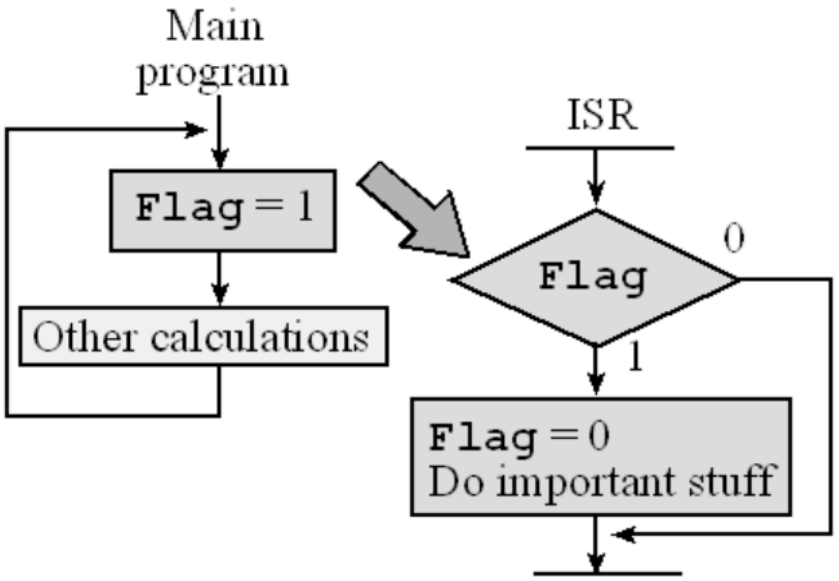
Binary Semaphore

- A binary semaphore protects access to shared resources using a shared flag.
  - One thread performs the signal action
    - Sets the flag.
  - One thread performs the wait action,
    - Waits for the flag to be set.
    - Clears the flag.
    - Performs shared data operations.
- The signal action can be in the ISR and the wait action in the lower priority thread.
- Example:
  - ISR is reading an ADC value.
    - ISR sets flag when new data is available.
  - main() is calculating the average.
    - Waits for the flag.
    - Reads the new ADC value into a local variable
    - Clears the flag.
    - Calculates the new average.



Binary Semaphore

- The signal action can be in the main function or lower priority ISR and the wait action in the real time ISR.

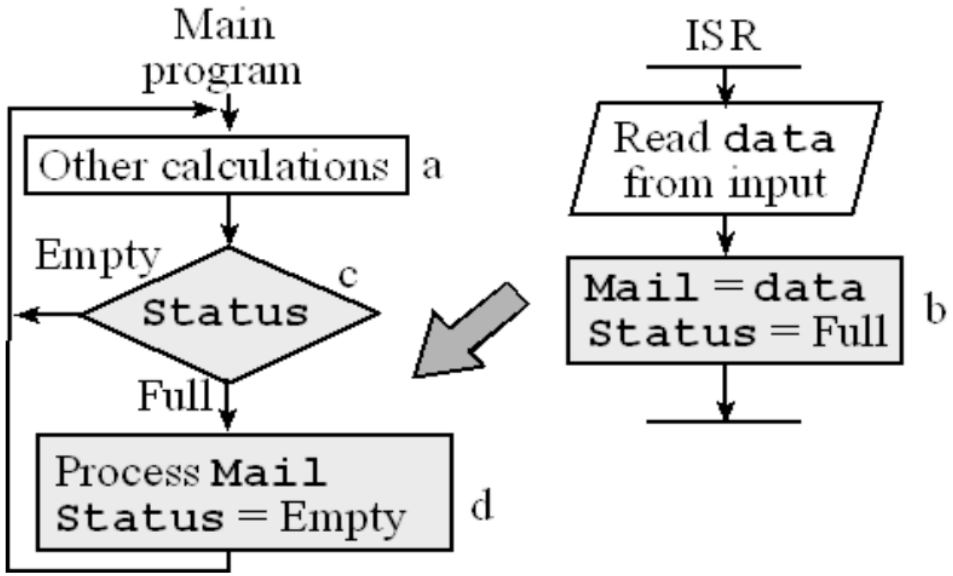


- Example:
  - ISR is responsible for reading a voltage monitor status.
  - main() detects an error in the status and sets the flag.
  - On the next ISR
    - The ISR clears the flag.
    - Reads the voltage status like it does every time but also reads the voltage itself to get more information on the failure.

Mailboxes

- Mailboxes are binary semaphores, that protect a specific data variable. They have two shared global variables.
  - Mail which contains the shared data.
  - Status which is the binary semaphore flag.
- The Mail can be a single variable, multiple variables, structures, etc. Whatever is required in the application.

- Sequence of events.
  - (a) main() is busy doing low priority functions.
  - (b) ISR functionality
    - Reads data from input.
    - Places data in Mail.
    - Sets Status = Full.
  - main()
    - (c) Polls Status.
    - If it is Full, reads the data from Mail into local variable.
    - (d) Set Status = Empty.



Mailbox: Lab0 Example

- Lab 0 has a shared data problem.
  - gButtons is read multiple times during the snprint statement for the LCD.
  - If a button press interrupt occurs while writing the button status string in main(). The wrong value could be displayed.
- Solution: Mailbox
  - Button ISR functions.
    - Waits for button press.
    - Button info is placed in gButtons (global variable).
    - Sets gMailboxStatus = 1. (global variable status = Full)
  - Main function polls status, if it is Full,
    - Reads gButtons into local variable.
    - Sets gMailboxStatus = 0. (global variable status = Empty)
    - Prints string using local variable.

Mailbox: Code Example

Example 2: Simple mailbox

```
int bMailboxFull = 0;
// arbitrary size mailbox data structure

void SomeISR(void) {
    // clear interrupt flag
    if (!bMailboxFull) {
        // put data into mailbox
        bMailboxFull = 1;
    }
}

void main(void) {
    // init
    while (1) {
        if (bMailboxFull) {
            // extract data from mailbox
            bMailboxFull = 0;
        }
    }
}
```

Define global Mail and Status variables

Initializes Status to Empty

ISR performs signal action

- If Empty, writes data to Mail and sets flag.
- If Full, no action taken.

main() performs wait action

- If Full, reads data and clears flag.
- If Empty, no action taken.

- What timing conditions between threads must be met for data not to be lost?
  - ?

Buffering

- Data can be buffered to avoid reading and writing to the same location.
  - Data is written to sequential locations in an array by one thread.
    - When the end of the array is reached, the index wraps around to index 0.
  - Data is read from the buffer by another thread.
  - The timing of the reads and writes are controlled to guarantee there is no overlapping access.
- Buffering (lab 1, ADC sampling routines)
  - High and low priority threads access different parts of the buffer.
  - ADC writing into one half of the buffer.
  - The oscilloscope function reads the buffer index and chooses to process the data in the part of the buffer not being written to.