Wireless Firmware Upgrades for the MSP430 Using a SPI Connected SoC

Kris Dickie – B.Tech
Clarius Mobile Health
kris.dickie@clarius.me

I. Introduction

The MSP430 is low-power microcontroller designed and produced by Texas Instruments, and come in a variety of models. The purpose of this technical paper is to explain how to perform a firmware upgrade over an external peripheral bus without relying on the MSP's on-board Bootloader (BSL). There may be many reasons to not use the BSL to perform a firmware upgrade, however the design referenced in this paper was put into place without addressing connected BSL pins for the sake of both simplicity and complacency. It is assumed that the reader has a good understanding of the MSP430 architecture, as concepts of ports, interrupts, and command execution are not explained in any great detail. It is also worth noting that all code was written in C/C++ and does not rely on assembly instructions. The program described also allocates no heap and makes static instantiations of C++ objects.

II. System Design

The electronics designed include a MSP430-5438A, a relatively flexible MSP model with a large address space and a variety of peripherals that can be addressed. It's primary purpose is to monitor activity over a UART bus connected to a Bluetooth Low Energy controller and respond to commands; one of which is to boot up a higher power-consumption ARM-based SoC, which is then used for other purposes such as enabling an 802.11ac wireless connection for high speed transmissions of compressed image data. The MSP controls the power pins to the SoC, and also shares a 3-wire SPI bus and two general purpose pins which are used for interrupts, one for interrupting the MSP from the SoC, and vice-versa. The interrupts are crucial to the design, as a SPI slave has no good mechanism to signal the master when it has data ready since it cannot control the clock. In this design, the SoC is the SPI master, and the MSP is the SPI slave. A different peripheral bus could be used as the communications port, such as I^2C or UART, however SPI was chosen for it's relative simplicity.

Throughout the lifetime of the operation of the device, software and firmware updates may come through the high-speed wireless connection, both for updating the SoC and the MSP. Since the design uses a battery, the device includes warning systems as to notify the user when a crucial segment of the software/firmware update is taking place as to ensure the battery is not removed and thereby 'bricking' the device.

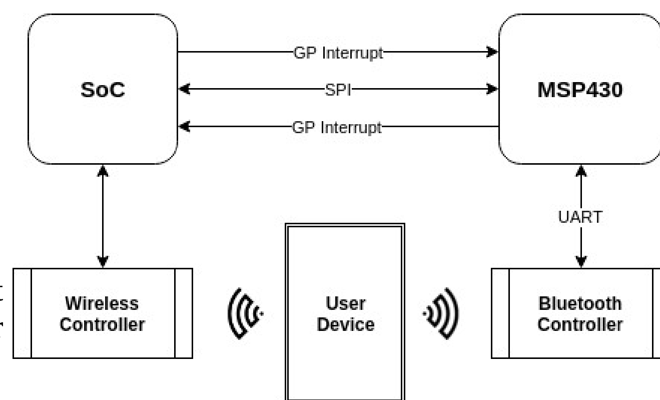

Figure 1. System Diagram

III. SPI Setup

As mentioned, the MSP is a SPI slave, and therefore cannot control the clock, and must send notifications through a separate GPIO pin to interrupt the SoC. Acting as a SPI slave on the MSP comes with some interesting nuances, that are

weakly documented in the TI User Guide's, but are crucial for robust operation. A SPI bus must *only* be setup on the MSP, when acting as a slave, once the clock is enabled by the SPI master, in this case once the SoC has powered up and setup it's peripherals. The MSP handles this by using the interrupt mentioned previously on one of the free pins, as a trigger that the SoC is alive, and it can now setup the SPI pins. An alternative that was tested on a development board is to also route the SPI clock to an interrupt pin, however this potentially wastes a pin and unnecessarily complicates routing. It's important to mention that on the MSP430-5438A, only Ports 1 and 2 can be used as general purpose interrupt receivers. To wait for a SPI interrupt on the peripheral would result in a chicken and egg scenario, and thus the above solution works quite elegantly.

Example:

```
bool spiReady = false;

#pragma vector=PORT1_VECTOR
void onPort1Interrupt()
{
  switch (P1IV)
  {
    // ex. schematic uses port 1.2 for
    // interrupt from SoC
    case P1IV_P1IFG2:
      if (!spiReady)
      {
        setupSpi();
        spiReady = true;
      }
      else
        handleInterrupt();
      break;
  }
}

// ex. schematic uses UCA2 for SPI
void setupSpi()
{
  // enable peripheral
  P9SEL |= (BIT0 | BIT3 | BIT | BIT5);
  // reset
  UCA2CTL1 |= UCSWRST;
  // CS, high polarity, MSB
  UCA2CTL0 |= (UCSYNC | UCMODE_2 |
               UCCKPL | UCMSB);
  // finish
  UCA2CTL1 &= ~UCSWRST;
  // enable rx interrupt
  UCA2IE |= UCRXIE;
}
```

The spiReady flag used in the above example should be reset when the MSP powers down the SoC, as to ensure the first interrupt always triggers a SPI bus configuration.

IV. SPI Communications

The implemented design makes use of numerous types of commands and data exchanges between the MSP and the SoC; thus a simple protocol was developed using a packet based system, where a packet consists of a single header byte, a command byte, a command info byte, a data byte, and two size bytes. The packet can then be followed by an arbitrary payload with a size as described in the header. The details of the protocol commands do not need to be described, but the background is important since this packet format is used when sending firmware upgrade data to the MSP.

```
// header for spi packets
typedef struct _spiCommHeader
{
  uint8_t tag_;
  uint8_t command_;
  uint8_t info_;
  uint8_t data_;
  uint16_t payloadSize_;

} spiCommHeader;
```

The MSP is burned in through a FET interface and upon successful bootup, the firmware hash is programmed into the MSP info flash memory via the SoC. During normal operation the MSP and SoC will communicate to operate the device, and when there is a software upgrade the embedded firmware hash in the upgrade package is compared to the hash stored on the MSP's info flash. If they do not match, the SoC can notify the user if they would like to perform an upgrade, and also check that battery levels are sufficient to sustain device power for the period of the upgrade.

V. The Upgrade Process

Once an upgrade has been confirmed by the user, many preparations happen on the device to ensure that both the SoC and MSP do not get interrupted. Wireless communication is shut down, including WiFi on the SoC and Bluetooth on the MSP; this is done by manually powering down the components through the respective software interfaces.

V.a. Parsing the Firmware Update

The firmware file comes in the form of a TI-TXT file, which is a hex-encoded text file that specifies base addresses for various streams of hex bytes. The format is very simple to parse and the SoC runs through the file, building up a stream of bytes with base address information that will be sent to the MSP.

To ensure that the MSP is fully erased, spaces between base addresses are padded with 0xFF, which is the byte value read back when a flash bank has been erased. Without doing this, stray instructions may still exist in flash, and although theoretically should never get executed, the precaution of mimicing the FET upgrade is taken.

A pseudo-code example for parsing the file looks similar to the following:

```
data = readTiTxt(filename)
while (readLine(data))
{
  if (find('@'))
  {
    addr = getStartAddr();
    if (bytesExist)
      padUntilNewAddr(0xFF);
  }
  else if (find('q'))
    padUntilEndOfFlashMem(0xFF);
  else
    addBytesReadFromLine();
}
```

A version without padding was also created, so that different 'blocks' were generated at various base addresses, however to get the full erase of the instruction flash, only a single block will be created with the code above, at a base address of 5C00H (where the MSP430-5438A instruction flash starts).

V.b. Setup to Execute from RAM

One of the more complicated, and definitely the most important piece of manually upgrading the MSP firmware is to execute instructions, and deal with interrupts, from RAM, as all of the instruction flash will eventually be erased and overwritten.

A custom linker file should be created for the MSP project so that the memory configuration can be split between general purpose execution and firmware upgrade execution. The project described allocates 1KB (400H bytes) of space in instruction flash starting at 5C00H, with a corresponding RAM space starting at 1C00H

An example linker excerpt is shown below:

```
MEMORY
{
  RAMSWU   : origin = 0x1C00, length = 0x0400
  RAM      : origin = 0x2000, length = 0x3C00
  FLASHSWU : origin = 0x5C00, length = 0x0400
  FLASH    : origin = 0x6000, length = 0x9F80
  FLASH2   : origin = 0x10000,length = 0x35C0
}

SECTIONS
{
  .bss        : {} > RAM
  .data       : {} > RAM
  .TI.noinit  : {} > RAM
  .sysmem     : {} > RAM
  .stack      : {} > RAM (HIGH)

  .flashswu   : load = FLASHSWU, run = RAMSWU
  .ramswu     : load = FLASHSWU

  .text       : {}>> FLASH2 | FLASH
  .text:_isr  : {} > FLASH
  .cinit      : {} > FLASH
  .const      : {} > FLASH | FLASH2
}
```

The RAMSWU and FLASHSWU sections are the address spaces that are used for exclusively for placing and executing instructions for performing firmware updates. It is important to note that the FLASHSWU section has a special *run = RAMSWU* indication, which forces all instructions

located within that section to run out of the RAM space that was allocated.

In the MSP program, some initial setup needs to be written to properly move instructions around, the first of which is to create a definition for the interrupt vector table. In our example, only two interrupts are used:

1. The SPI interrupt on USCI A2
2. The general purpose interrupt on Port 2

A structure similar to below should be setup so that pointers to functions can be setup.

```
// isr function pointer variable type
typedef uint16_t* isr_type_t;

// interrupt vector table data type
typedef struct _isrVect
{
  isr_type_t reserved[41];
  ....
  isr_type_t io_p2;
  ....
  isr_type_t usci_a2_rx_tx;
  ....
  isr_type_t reset;

} isrVect;
```

Some static variables can be setup and initialized when the MSP boots up, or addresses can be hard-coded, for readability, excerpts from the actual source code are found below:

```
// define important addresses / sizes
#define FLASHSWU    0x5C00
#define RAMSWU      0x1C00
#define TOP_OF_RAM  0x5BFF
#define SWU_SIZE    0x0400

// start address of interrupt vector
// table in RAM
#define IV_START_ADDR /
  (TOP_OF_RAM + 1 – sizeof(isrVect))

isrVect* iv = (isrVect*)(IV_START_ADDR);
uintptr_t flashSwu = FLASHSWU;
uintptr_t ramSwu = RAMSWU;
```

Casting of functions can be interesting when trying to assign a function pointer to the interrupt vector table, so a special type definition was created.

```
typedef void (*ivFn)();
```

When ready to move the flash instructions into RAM for execution, a simple memory copy will do.

```
uint8_t* ram = (uint8_t*)ramSwu;
uint8_t* flashmem = (uint8_t*)flashSwu;
memcpy(ram, flashmem, SWU_SIZE);
```

To move the interrupt vector table into RAM requires some special casting as mentioned above since we are using a large code model (20 bit pointers), and the ISR vector must be 16 bit pointers. Note that once this happens, there is no turning back, and ALL interrupts will be handled through the newly loaded interrupt vector table!

```
// handler for spi bus interrupts
ivFn fn1 = onSpiData;
void* ptr = *(void**)(&fn1);
iv->usci_a2_rx_tx =
  (uint16_t*)(*(uint16_t*)(&ptr));

// handler for gp interrupts
ivFn fn2 = onP2Interrupt;
ptr = *(void**)(&fn2);
iv->io_p2 =
  (uint16_t*)(*(uint16_t*)(&ptr));

// move!
SYSCTL |= SYSRIVECT;
```

That's it. Now everything is ready to execute in RAM; the next tricky part is to ensure that ALL execution and variable access happens within RAM and not in the now very volatile flash that will be overwritten during the firmware update.

V.c. Writing Functions to Execute in RAM

Once the interrupt vector table and the appropriate instructions have been loaded into RAM, it will be important to ensure that ONLY these loaded instructions execute and that ALL data access happens within RAM.

The typical functions that are required to to execute

during the firmware update include:

- The SPI interrupt handler
- The general purpose interrupt handler
- Parsing functions when receiving data from the SPI bus
- Functions to write to the instruction flash memory
- A no operation handler

The simple way to put a function into the proper flash/RAM space is to put the following pragma command directly before the implementation of the function:

```
#pragma CODE_SECTION(".flashswu")
```

This ensures that the instructions are in the FLASHSWU section, and that they get copied into RAM when the memcpy() command is executed as shown in the previous section. Note this works for Code Composer Studio (CCS), but IAR has a different syntax for moving code into specific sections.

A no-operation (nop) handler is important since most programs will enter a low power mode, and have a main loop which becomes idle at some point. An interrupt will then wake up the MSP and the main loop can then then process interrupt data. Since the general purpose execution must cease to exist, running a nop loop is the safest way to ensure nothing gets executed outside of RAM, for example:

```
#pragma CODE_SECTION(".flashswu")
// loops continuously while executing
// in ram
void nop()
{
  while (1)
    __no_operation();
}
```

For C++ classes, it may be cumbersome to constantly ensure that functions are placed into RAM through pragmas, so one simple method to ensure execution from RAM is to create inline functions, assuming that they are not inherently large. This can be useful to run a simple function to toggle an LED for sample.

V.d. Data Access From RAM

As mentioned in the previous sections, it is imperative that all data access, whether constant or variable is done on variables that exist in RAM or on the stack. When writing C code, it is pretty common that most global variables automatically become static, and are therefore automatically assigned to RAM, however with C++, class members for example may be assigned to flash. It is important to look at the mapping file generated after compilation to try to understand where various variables and functions have been assigned to.

For C++ classes, ensuring that variables exist in RAM can be simply done by using the static keyword.

V.e. Firmware Data Exchange

Once the bytes have been loaded into SoC memory after parsing the TI-TXT file, they are then sent over the SPI bus to the MSP in 512 byte chunks. 512 bytes was chosen because segment erases, which are the smallest flash erases that can take place, are also aligned at 512 bytes.

The SoC will send setup information first, which describes the base address the set of bytes should be written to, and how many bytes will be written. After the setup information, actual instruction data will be sent.

On the MSP side, the SPI interrupt vector that is newly loaded into RAM will occur once the SoC writes it's commands, and parsing will occur, for example:

```
#pragma CODE_SECTION(".flashswu")
__interrupt void onSpiData()
{
  switch(__even_in_range(UCA2IV, 4))
  {
  // rx data
  case 2:
      parse(UCA2RXBUF);
      break;
  }
}
```

The parsing function will handle setup of data, actual instruction data, and a reboot command that is sent by the SoC once all instruction data has been sent.

```
#pragma CODE_SECTION(".flashswu")
void parse(uint8_t b)
{
  if (parseByte(b))
  {
    uint8_t cmd = parsedCommand();
    switch (cmd)
    {
      // sent by SoC to complete
      // software update
      case swuReboot:
        PMMCTL0 = PMMPW | PMMSWBOR;
        break;
      // sent by SoC to setup
      // address, size, etc.
      case swuSetup:
        setupWrite();
        notifySoC();
        break;
      // new instruction data sent
      // by SoC (512 bytes)
      case swuData:
        writeData();
        notifySoC();
        break;
    }
  }
}
```

The code to write to flash is relatively simple and for the firmware update, the function analyzes the data sent to see if it is padded for an erase, and then skips the write to save on write time.

```
#pragma CODE_SECTION(".flashswu")
void writeToFlash(uint32_t addr,
     const uint8_t* data, uint32_t sz)
{
  uint32_t i;
  bool blank = true;

  __disable_interrupt();
  // erase segment
  FCTL3 = FWKEY;
  FCTL1 = FWKEY + ERASE;
  __data20_write_char(addr, 0);
  while (FCTL3 & BUSY);

  // check if the block is blank
  // if so, then skip the write,
  // and just be happy with the erase
  for (i = 0; i < sz; i++)
  {
    if (data[i] != 0xFF)
    {
      blank = false;
      break;
    }
  }
  if (blank)
  {
    FCTL3 = FWKEY + LOCK;
    __enable_interrupt();
    return;
  }

  // enable write
  FCTL1 = FWKEY + WRT;

  // write data
  for (i = 0; i < sz; i++)
  {
    __data20_write_char(addr + i,
                        data[i]);
    while(!(FCTL3 & WAIT));
  }

  // disable wrt
  FCTL1 = FWKEY;
  while (FCTL3 & BUSY);
  // set lock bit
  FCTL3 = FWKEY + LOCK;
  __enable_interrupt();
}
```

It's also worth mentioning that since we have a 20-bit address space in the MSP430-5438A, the __data20_writeXXX functions are used as opposed to a simple address assignment. This ensure that any instruction outside of the 16-bit address space will actually get written.

VI. Conclusion

The MSP430 is a flexible low-power MCU that can perform very well in conjunction with a higher power SoC to operate a complex device. Since today's developed devices are often out of date tomorrow, it is important that over-the-wire or wireless (think IoT) updates can happen without JTAG or FET connections that require custom servicing of the device itself. If MSP430 upgradeability becomes an afterthought in the design process, all is not lost, as long as a bus exists that allows for two-way communications.

As shown, the MSP430 can be re-programmed to force execution out of RAM to completely overwrite it's instruction flash with relative simplicity and ease.

The author intends to create a GitHub repository with various code examples that were briefly shown here. A subsequent version of this paper will be published with any updates and a repository link.

About the Author

Kris studied Computer Systems at the British Columbia Institute of Technology, and has since been working in the medical device industry for over 16 years, helping to design and implement real-time imaging devices using a variety of platforms and technologies, which include: Microsoft Windows, Embedded Linux (ARM based), TI C6X DSPs, TI MSP430, ADI Blackfin SoC, and Xilinx FGPAs.