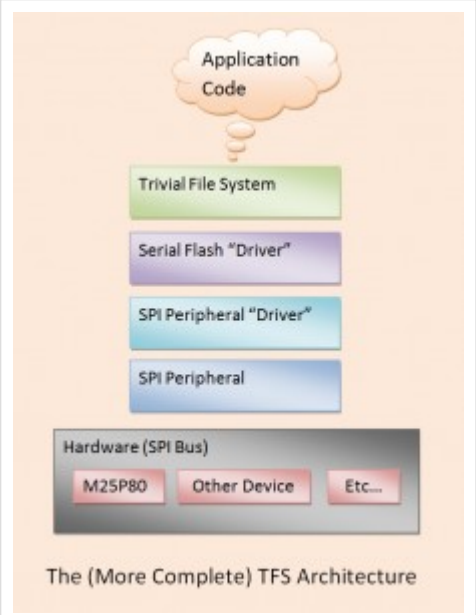
Bus Management

Posted on **June 3, 2012**

One side of the M25P80 abstraction that has been omitted until now is the interface layer between the processor and the device, namely the Serial (or sometime Synchronous) Peripheral Interconnect (SPI) bus. SPI is a common serial peripheral bus whose use in embedded systems can be traced back to the 1980s.



To the left is a much more complete architecture diagram. Added to the picture are the lower levels, in particular the SPI bus, peripheral and driver levels.

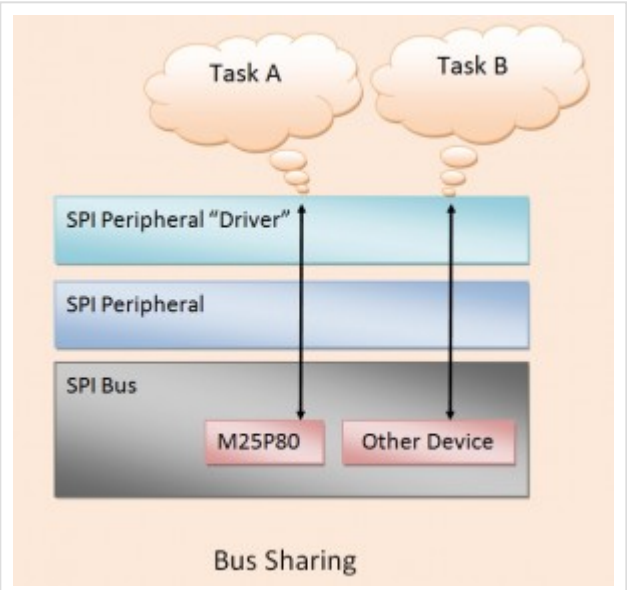
Starting from the bottom, it must be made clear that the SPI bus supports one controlling node (sometimes called the master node) and one or more peripheral nodes (sometimes called slave nodes). In general, the controller selects a peripheral and then the two exchange data. When the exchange is completed, the controller deselects the peripheral at which time another peripheral node may be selected.

In connecting the M25P80 to the PIC, some decisions need to be made. In particular, which of the two available SPI ports is to be used and to which I/O pin will the chip select be connect. In this case, both of those decisions were made by the engineers at MikroElektronika who designed the MMB32 board being used in these examples. Of course the M25P80 driver software has to comply with these design specs. Now the required selections could just be hard coded but that would make the code hard to re-use, so instead, all of the hardware dependent “settings” are brought into a single header file “MikroeMMB32.h”. The following little bit of that file deals with the M25P80:

```
259  // Serial Flash Memory Chip Select.
260  #define FLASH_CS_BIT_MASK    (1 << 2)
261  #define FLASH_CS_TRIS_CLR    TRISCCLR
262  #define FLASH_CS_TRIS_SET    TRISCSET
263  #define FLASH_CS_LAT_CLR     LATCCLR
264  #define FLASH_CS_LAT_SET     LATCSET
265
266  // Serial Flash Memory SPI port.
267  #define emTakeFlashSPI       emTakeResSPI_2
268  #define emGiveFlashSPI       emGiveResSPI_2
269  #define emQryFlashSPI        emQryResSPI_2
270  #define emSetFlashSPI        emSetResSPI_2
271  #define FLASH_SPI_CHANNEL    SPI_CHANNEL2
```

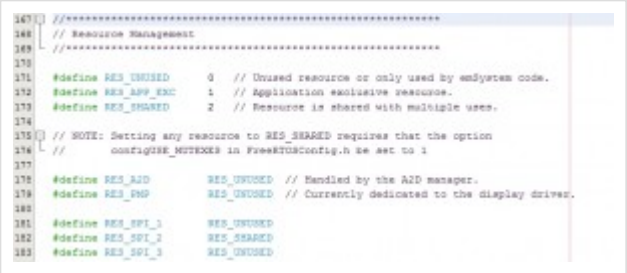
The first define is used to specify the bit used as the device chip select, the next two define the registers needed to control the direction of that bit, and the two after that for actually controlling the state of the chip select bit; active (low) or inactive (high). Note that in each case, rather than specify the appropriate TRIS or LATCH register directly, the code specifies the respective clear and set registers so that these operations may be performed “atomically”. Now, this code is still a work in progress, so I know the I/O control could stand a great deal more in the abstraction area, but for now, this works OK. (See the note at the end)

Skipping to the last line, the FLASH_SPI_CHANNEL define specifies which SPI port the device will be using. Simple. So what of the rest? They exist to handle the fact that SPI is potentially a multi-device bus, that only supports one operation at a time. This rule can be enforced a number of ways; for example only one task might be allowed to access the SPI port, preventing any possible conflict.



If multiple devices need to be accessed by multiple tasks (as is illustrated), what is needed is called a Mutual Exclusion or Mutex semaphore. For now, a Mutex is used to prevent multiple tasks from accessing a resource out of turn. (See also below)

In order to protect the SPI port from inappropriate accesses, it is first needed to define how that port will be accessed. This is done in the header file “Confiig.h” The following little bit of that file specifies the use of the various system resources:



Note how RES_SPI_2 is specified as RES_SHARED whereas others are specified as RES_UN–USED. This prevents wasting resources by not creating Mutexes for unused or unmanaged peripherals. This is then used back in the header file “MikroeMMB32.h” to create macros for managing the Mutex and device setup if it is shared, or doing nothing, if the device is NOT shared. This way, the code is not cluttered with even more #if … #endif blocks than needed and again, one file contains the definitions if they need to be updated. The result of this is two simple macros: emTakeFlashSPI and emGiveFlashSPI. To transparently share the SPI port, the following code template is employed:

```
emTakeFlashSPI();
// code use the Flash's SPI port here.
emGiveFlashSPI();
```

Finally, it is necessary to account for the fact that different tasks may use different settings of the SPI port. This is handled by the emQryFlashSPI and emSetFlashSPI macros. These query and set a configuration byte that tracks the setting of the SPI port. emQryFlashSPI returns true if the configuration does not match the one specified, indicating that the port needs to be set up. An example of this in action is:

```
if (emQryFlashSPI(emSPI2_M25P80))   // Confirm the configuration.
{
    // Close any previous configuration.
    if (emQryFlashSPI(emSPI2_NONE))
        SpiChnClose(FLASH_SPI_CHANNEL);

    // Set up the M25P80 configuration.
    // SPI2 setup
    SpiChnOpen(FLASH_SPI_CHANNEL,
               SPI_OPEN_MSTEN | SPI_OPEN_MODE8,
               PB_FREQ/bitRate);
```

```
    // Set the config indicator.
    emSetFlashSPI(emSPI2_M25P80);
}
```

The crucial thing about these macros is that they only result in code being generated if the SPI port is actually shared. If the port is not shared, they translate to no code at all, or “non-executed code” like if (0) { … } that is easy for the compiler to eliminate. Thus the burden at run time is minimized without hard coding system details into the “driver” code.

This is perhaps one of the most complex postings to date, and I must admit, it has left a lot of loose ends that will need to tied up in future posts. (See below once more)

As always, comments and suggestions are welcomed and invited.

Peter Camilleri (aka Squidly Jones)

PS: I should also add that another reason that the I/O are not as abstracted as they could be is that I am waiting to see what Microchip does in the next generation of peripheral libraries for the PIC32 “C” compiler. These were promised at the 2011 Masters Conference and to be honest, I’ve not heard anything more on that front since.

PPS: A future article looking at RTOS issues called FreeRTOS Impressions is planned. Please bear with me while it is being worked out.

PPPS: All of this software exists within a framework dubbed emSystem, hinted at in the article About the MMB-32 development project. A future article will begin a more formal look at the emSystem, its design and architecture.

SHARE THIS:

Print LinkedIn Facebook Twitter

LIKE THIS:

Loading...

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=851]** .

1 THOUGHT ON “BUS MANAGEMENT”

Appliance repair Beverly Hills ca

on **June 20, 2012 at 1:58 am** said:

Congratulations on having one of the most sophisticated blogs I've came across in some time! It's just incredible how much you can take away from something simply because of how visually beautiful it is. You've put together a great blog space great graphics, videos, layout. teuthida-technologies.com is definitely a must-see blog!

Comments are closed.