January 2021

www.ccsinfo.com
262-522-6500

<BITS & BYTES> Newsletter

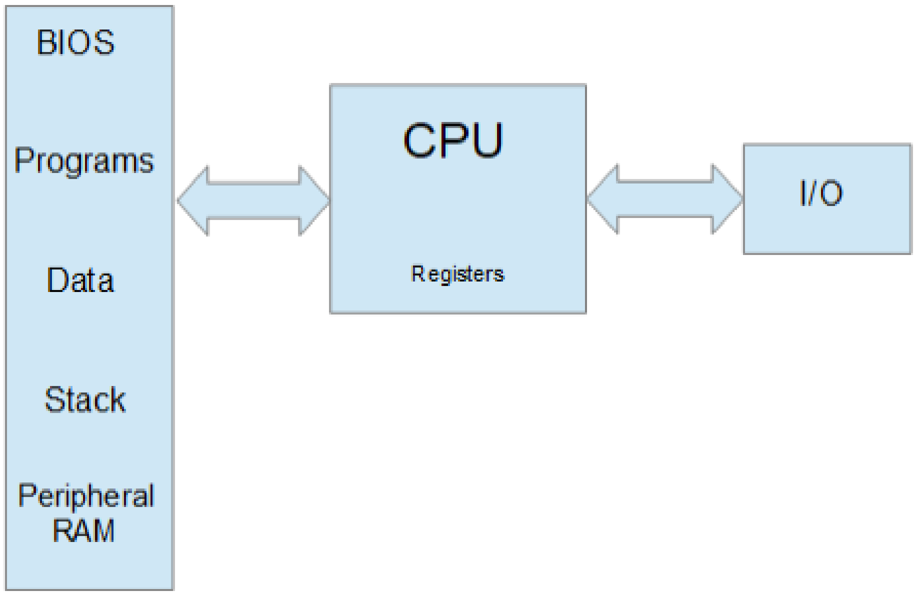
PIC24FJ
Development Kit

The development kit contains everything you need to begin development with Microchip's PIC24FJ family.

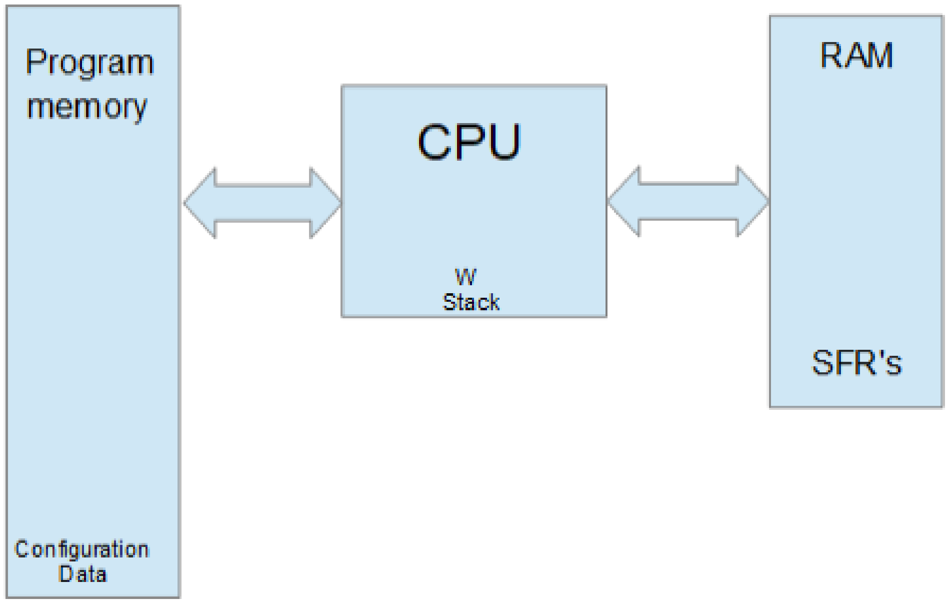
INSIDE THIS ISSUE:

PIC® MCU Address Spaces- Part 1

A common question asked is why code written for a PC cannot be used on microcontrollers. When running a program on a x86 style PC, the program is loaded into memory and begins executing. When that program needs data, it allocates more memory in the same RAM address space. Even the ROM BIOS, video RAM and more, is all in the same address space differentiated only by its address. The PC does have a second separate address space for I/O usually only used to communicate with peripherals.



On a PIC® MCU the memory is different. There are two buses, one for program memory and the other for RAM, peripherals and special registers. That means there are two address 0’s. A 0 in program memory and the other in RAM. This is referred to as a Harvard architecture.



Note that the buses on each side of the above diagram are different data width sizes. On the left, is 12,14,16 or 24 bits depending on the PIC® MCU. On the right it is 8 or 16 bit. One advantage to this architecture is both memory spaces can be accessed at the same time.

Look at the following C:

```
char s[10];
strcpy(s,"Hi There");
```

On a PC the **"Hi There"** is loaded with the program into memory. The **s** is allocated to another area of the RAM. For example if **"Hi There"** starts at location 0x1108 and s is allocated at 0x2004. The function call looks like this:

```
strcpy(0x2004, 0x1108);
```

On a PIC® MCU, by default the "Hi There" is in program memory, such as 0x108. **s** is in the RAM address space like maybe 0x34. The call would look like this:

```
strcpy(0x34, 0x108);
```

The problem is how does **strcpy**() know if 0x108 is in the program memory space or RAM space? Remember there is a 0x108 address in both.

This problem can come up whenever a pointer is used. In the CCS C compiler the following syntax is used to declare pointers. ROM refers to the left side memory above and RAM to the right side.

`char id;`	id is stored in RAM
`char * id;`	id is stored in RAM, is a pointer to RAM
`rom char id;`	id is stored in ROM
`char rom * id;`	id is stored in RAM, is a pointer to ROM
`char * rom id;`	id is stored in ROM, is a pointer to RAM
`rom char * rom id;`	id is stored in ROM, is a pointer to ROM

Consider this declaration:

```
char * a;
char rom * b;
```

When the compiler encounters *a in the code it uses the address in **a** and grabs data from the RAM address space (right side). When the compiler encounters ***b** in the code, it uses the address in **b** and grabs data from the ROM address space (left side). The following would cause nothing but trouble:

```
a=b;
```

For function calls the parameters act like assignments so care must be taken not to mix pointers from different address spaces. The same kind of care is not needed on some other architectures like PC's and that is why some ported code does not work.

Back to the strcpy() the reason it works in the CCS compiler is there are two overloaded functions defined like this:

```
strcpy( char * dest, char * src);
strcpy( char * dest, char rom * src);
```

In the above example, the second function was called. Not all compiler functions have multiple versions defined. For example, there is only one version of strcat() defined, for RAM only pointers. This can be annoying if your code uses both RAM and constant strings in function calls. One way around this is to always pass RAM strings like this:

```
write_to_log(char * string);
                    ...
char temp[10];
strcpy(temp, ”Hi There”);
write_to_log(temp);
```

The compiler has a feature where you can tell it to always copy constant strings to a temporary RAM area and then pass the RAM pointer. Here is how it looks:

```
#device  pass_strings=in_ram
     ...
write_to_log(char * string);
     ...
write_to_log(”Hi There”);
```

Users need to be careful because the same RAM area is used the next time a function call is made where it needs it, so the pointer is only valid until the next call.

For some users the **rom** keyword above may seem new. A common approach to putting data in rom is like this:

```
const char message[10] = ”Hi There”;
```

By default, the compiler saves message in ROM, however, the format is not straightforward. Depending on the chip, the format is different so the data can be saved in the most efficient way possible. That saves memory, however users can not create pointers to **const** data. In the above users can do **message[i]** but not **&message**.

This can cause trouble porting code from other architectures so the compiler provides an alternative interpretation of **const**. In ANSI C **const** data is in RAM and the compiler throws an error if you try to change the data. To get that in CCS C do this:

```
#device  const=read_only
```

As for the default **const** data the format varies depending on the chip and sometimes the data itself. For example, on some chips the assembly equivalent of this is used:

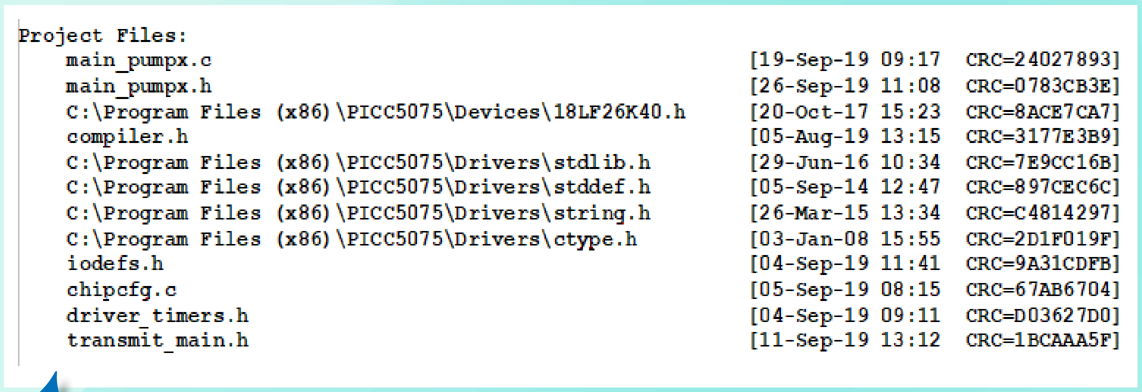
```
char lookup_const123(int index) {
   switch(index) {
     case 0 : return ‘H’;
     case 1 : return ‘i’;
     case 2 : return ‘ ‘;
     case 3 : return ‘T’;
     case 4 : return ‘h’;
     case 5 : return ‘e’;
     case 6 : return ‘r’;
     case 7 : return ‘e’;
     case 8 : return 0;
   }
}
```

On a 14 bit PIC® MCU, if the constant data is all under 128, then two items can be packed into each 14 bit word. On a 24 bit part we can pack in 3 bytes of data in each word. None of these methods can provide an ANSI compliant pointer. When the **rom** is used, the data is always saved with one item per pointer value so pointers are fully supported. Users should be aware on 24 bit parts rom also gets packed three bytes per instruction. Instructions take two addresses. The compiler does an internal translation from a byte address to the device address.
Be aware, although the pointer works as it should, users should not expect to use the pointer for something other than allowing the compiler to access the data.

This article covered the two major address spaces used by the CCS C compiler. The compiler also allows for user defined address spaces. That will be covered in detail in part 2 of this article. Part 3 will cover PIC® MCU chips that have multiple ways of configuring their address space.

CCS Inc COMPILER FEATURE FOCUS

Do you know the CCS compiler symbol file shows all source files used for a build along with the CRC's of the file?



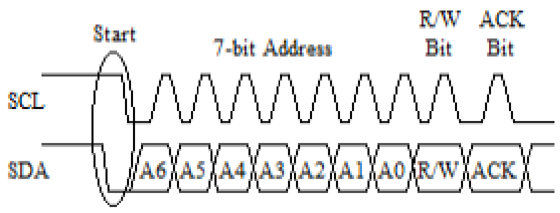
```
Project Files:
    main_pumpx.c                                            [19-Sep-19 09:17  CRC=24027893]
    main_pumpx.h                                            [26-Sep-19 11:08  CRC=0783CB3E]
    C:\Program Files (x86)\PICC5075\Devices\18LF26K40.h     [20-Oct-17 15:23  CRC=8ACE7CA7]
    compiler.h                                              [05-Aug-19 13:15  CRC=3177E3B9]
    C:\Program Files (x86)\PICC5075\Drivers\stdlib.h        [29-Jun-16 10:34  CRC=7E9CC16B]
    C:\Program Files (x86)\PICC5075\Drivers\stddef.h        [05-Sep-14 12:47  CRC=897CEC6C]
    C:\Program Files (x86)\PICC5075\Drivers\string.h        [26-Mar-15 13:34  CRC=C4814297]
    C:\Program Files (x86)\PICC5075\Drivers\ctype.h         [03-Jan-08 15:55  CRC=2D1F019F]
    iodefs.h                                                [04-Sep-19 11:41  CRC=9A31CDFB]
    chipcfg.c                                               [05-Sep-19 08:15  CRC=67AB6704]
    driver_timers.h                                         [04-Sep-19 09:11  CRC=D03627D0]
    transmit_main.h                                         [11-Sep-19 13:12  CRC=1BCAAA5F]
```

Go To COMPILE->SYMBOLS to view the .SYM file.

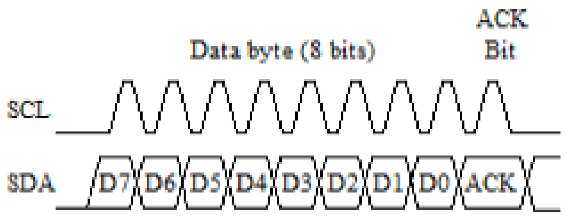
This helps with solving problems like builds on one PC that differ from another PC or ensuring the source code that was used for a given release. Be sure to save the .SYM file with your sources to make the most use of this feature.

The I²C protocol is a very useful serial communication protocol to communicate with multiple devices using only two I/O pins of the PIC® MCU, the SCL and SDA pins. The SCL pin is the clock pin used to synchronizing the clocking of data into and out of the devices, and the SDA pin is the data pin that the data is clock into and out of the devices. Both pins are open drain configured, meaning that to output a low on one the of pins, the pin is driven low and to output a high on one of the pins, the pin is floated, requiring an external pull-up resistor to pull the line to the high voltage level.

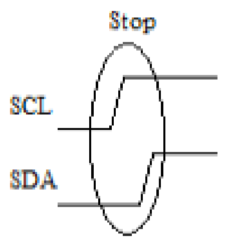
All I²C communication is started by the master device by it doing an I²C start on the bus, pulling the SDA pin low while both pins are high and then pulling the SCL pin low. Generally there is only one master device on the bus, however there is a multi-master version of the I²C protocol which is beyond the scope of this article. Next the master clocks out the address bits, there is a 7-bit address and 10-bit address protocol, 7-bit addresses are typically used in microcontroller applications so this article will focus on them. After the 7-bit address has been clocked, the read/write bit is clocked out by the master. If the read/write bit is pulled low the master is writing data to the slave and if it is high, the master is reading data from the slave. Finally following the read/write bit the master will clock in the acknowledge (ACK) bit. If a slave device acknowledges the address it will pull the SDA pin low during the ACK bit, or if no slave device acknowledges the address the SDA pin will be left to float high during the ACK bit. The following is a diagram showing what the I²C start and address bytes look like:



Assuming a slave device acknowledges the address byte the master will then clock out or clock in the data bytes. This is done by the master clocking 8 data bits followed by an ACK bit. If the master is writing data to the slave device, the master controls the SDA pin during the 8 data bits and the slave controls the SDA pin during the ACK bit. If the slave acknowledges the data byte it will pull the SDA pin low during the ACK bit and if it does not acknowledge the data byte, it will allow the SDA pin to float high during the ACK bit. If the master is reading data from the slave device the slave controls the SDA pin during the 8 data bits and the master controls the SDA pin during the ACK bit. The master uses the ACK bit during reads to signal the slave device if it will be reading more data or not, the master pulls the SDA pin low during the ACK if it has more data to read from the slave device and lets the SDA pin float high during the ACK if it done reading data from the slave device. The following is a diagram showing what the I²C data read and write bytes look like:

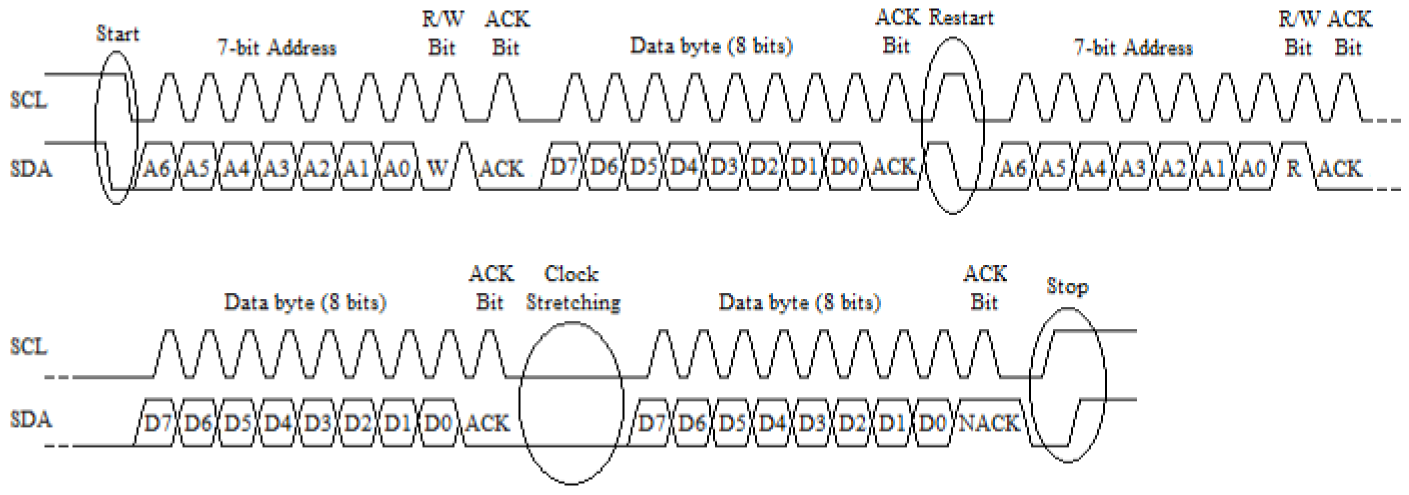


When the master is finished read or writing data it does an I^{2}C stop to indicate that it is done communicating with the device. The master should also do an I^{2}C stop any time the slave device does not pull the SDA line low during the ACK bit when the slave device is controlling the SDA line. The I^{2}C stop is done by transitioning the SDA line from a low to high when the SCL line is already high. The following is a diagram showing what the I^{2}C stop looks like:



A couple other features to know about the I^{2}C protocol is and I^{2}C restart or repeated start and clock stretching. The I^{2}C restart is done by the master first setting the SDA line high then letting the SCL line go high and then pulling the SDA line low again. The restart is used in cases were a master is reading or writing data to or from a slave device and then wants to switch the operation it is doing for the same slave device. An example of when a restart may be used is when reading data from a random address from an external EEPROM, the master would first write data to the EEPROM to set the address in the EEPROM memory it wants to read from and then do a read to read the data from that address.

Clock stretching is used by the slave device to pause the master from clocking data to or from the slave device. The slave device preforms clock stretching by pulling the SCL line, when it ready for the master to resume clocking it will release the clock allowing it to float high. An example of when clock stretching may be used is when multiple bytes of data are being read from a slave device, the slave device may need to pause the master so it can load the data to be read before the master starts trying to clock it out. The following is a diagram depicting theses two features:



The CCS C Compiler has a built-in library for doing I^{2}C communication, that library is capable of doing both a SW implementation or using the PIC® microcontroller's HW peripheral for doing the communication as well as being a Master or Slave device.

Slave mode is only supported when using the PIC® microcontroller's HW peripheral. Most PIC10, PIC12, PIC16 and PIC18 devices have one or in some case two Synchronous Serial Port (SSP) modules which can be used for I^{2}C communication. The SSP module is a combined SPI and I^{2}C peripheral that can be configured to do either SPI or I^{2}C communication. The CCS C Compiler's #use library(), when the correct parameters are passed to it, will setup the SSP modules for I^{2}C communication. When setup as a Master device the i^{2}c_start(), i^{2}c_write(), i^{2}c_read() and i^{2}c_stop() are used to perform the I^{2}C communication. When setup as a Slave device the SSP interrupt and i2c_isr_state(), i2c_read() and i2c_write() functions are used for the I^{2}C Communication, see ex_slave.c for an example of setting up slave communication. The following is an example showing how a Master would read a random byte from an external EEPROM:

```
   #use i2c(I2C1, MASTER, fast, stream=EEPROM_STREAM)

   int8 rAddress;
   int8 rValue;

   i2c_start(EEPROM_STREAM);
   i2c_write(EEPROM_STREAM, 0xA0);
   i2c_write(EEPROM_STREAM, rAddress);
   i2c_start(EEPROM_STREAM);
   i2c_write(EEPROM_STREAM, 0xA1);
   rValue = i2c_read(EEPROM_STREAM, 0);
   i2c_stop(EEPROM_STREAM);
```

Some newer PIC18 devices, the PIC18F47K42 family for example, have a new dedicated I^{2}C peripheral for doing I^{2}C communication. When setting up one of these new devices as a Slave, everything is basically the same the only exception is the ISR that is used, see ex_i^{2}c_slave_k42.c for an example. When setting up one of the these new devices as a Master on the other hand things are different. The reason of this is that the new dedicated I^{2}C module works differently then the SSP module did. The main difference is that old SSP module had individual bit for doing a start, restart, stop, etc. The new dedicated I^{2}C module do not have these, instead there are registers to set the address to send set the number of bytes transfer, whether is writing or reading data, etc. All the I^{2}C communication is handled by the peripheral, doing the start, restart, stop, etc. Because of this the CCS C Compiler legacy I^{2}C function are not compatible with these device when setup as a Master and using the HW peripheral. So in order to use the HW I^{2}C peripheral on these devices, the i^{2}c_transfer(), i^{2}c_transfer_out() and i^{2}c_transfer_in() functions were added to the #use i^{2}c() library. The i^{2}c_transfer() function is used to both write and read data to and from a slave device, doing a restart between the write and read. The i^{2}c_transfer_out() function can be used to write data to a slave device, and the i^{2}c_transfer_in() function can be used to read data from a slave device.
See ex_i^{2}c_master_hw_k42.c for an example of their use. Also these new functions were made compatible with all other PIC® microcontrollers both when using a software implementation or using the HW peripheral, so code written using these new functions will work on all PIC microcontrollers. The following are examples showing how a Master would read a random byte from an external EEPROM using the i^{2}c_transfer() functions:

```
#use i2c(I2C1, MASTER, fast, stream=EEPROM_STREAM)

   int8 rAddress;
   int8 rValue;

   i2c_transfer(EEPROM_STREAM, 0xA0, &rAddress, 1, &rValue, 1);
            //Does - Start, Write, Restart, Read and Stop
```

```
// or

   i2c_transfer_out(EEPROM_STREAM, 0xA0, &rAddress, 1);
             //Does - Start, Write and Stop
   i2c_transfer_in(EEPROM_STREAM, 0xA0, &rValue, 1);
             //Does - Start, Read and Stop
```

PIC24, dsPIC30 and dsPIC33 devices have a dedicated I^{2}C module for doing the I^{2}C communication. Their dedicated I^{2}C module is similar to the SSP module and all the function used for the master and slave communication are the same, including the the new i^{2}c_transfer() functions. The only exception to this is the ISR that is used for slave communication, see ex_slave_pcd.c for an example of slave I^{2}C communication on these device.

Bundle up and get back into coding projects with a C Compiler

$25 Off
a full Compiler
or Compiler Maintenance

USE CODE:
Winter2021

COVID-19 RESPONSE

During this time of global uncertainty and change, we want to assure you that we are taking every precaution to ensure that we can safely support our customers during this time.

Despite these challenges, CCS staff is continuing to provide technical support, as well as processing orders. It is essential customers have the tools they need to provide the development of existing or new products that may be necessary in the fight of Covid-19.

Many of our existing customers are having to work from home and we want to remind everyone of our Software Licensing Agreement. We pre-register all compilers in a user's name. You can install your compiler on your home PC and laptops. If you do not have access to the registration files and installer, contact customer service for assistance.

Most importantly, as we work together in this unique and rapidly changing environment, we do so with confidence that we will overcome this challenge. Until then, we hold our enduring commitment to the health and well-being of our employees and customers.

Please let us know how we can help you. Stay healthy.

More than 25 years experience in software, firmware and hardware design and over 500 custom embedded C design projects using a Microchip PIC® MCU device. We are a recognized Microchip Third-Party Partner.

www.ccsinfo.com

Follow Us!