

Using the PIC18F4620 or PIC18F4520

Note that there are some missing steps in this documentation

I am making corrections. and will post them shortly.

Historically, MIOS was developed to run on a core module stuffed with a PIC18F452. Recently, the PIC18F4620 has become available. It is near code-compatible with the 452, but features a significant increase in RAM/EEPROM/Codespace. See [the PIC18F4620 page](#) for details.

The following are intructions on converting old apps, and developing new apps, to run on the PIC18F4620. Small changes to the procedure make it compatible with the PIC18F4520 also.

OS Layers

MIOS v1.9b or above is required. You will need to download the MIOS source from [The uCApps.de Download Page](#) or [Directly](#). I recommend checking the first link for the latest version, as the '4620 is current in beta.

The Bootloader and MIOS recompile steps which follow should not be necessary for most cases of '4620 use, as these components are now available precompiled and packaged in a zip file [hosted on uCApps.de](#) Instructions follow for reference only, or for '4520 use.

Bootloader

Bootloader v1.2, which is packaged with MIOS v1.9 and up, will need to be recompiled as follows:

- Extract the MIOS source files from the zip
- Edit bootloader\main.asm
- Change

```
#define PIC_DERIVATIVE_TYPE 0
```

To

```
#define PIC_DERIVATIVE_TYPE 1
```

- Compile the project  [wiki link](#)

- Burn the hex file to the PIC  wiki link

MIOS



The MIOS Operating System itself must also be compiled, as follows:

- Edit src\mios.h from the MIOS source files
- Change

```
#define PIC_DERIVATIVE_TYPE 0
```

To

```
#define PIC_DERIVATIVE_TYPE 1
```

- Compile the project  wiki link
- Upload the hex file with MIOS Studio  wiki link

Please note that the above instructions should work for PIC18F4520 also. The only difference is that the PIC_DERIVATIVE_TYPE should be '2', not '1'. This stands for all of the following instructions.

Application Layer

Once your PIC18F4620 has the Bootloader burned onto it, and MIOS uploaded, you are ready to upload your application. A few modifications may be required:

Migration

If you have an existing ASM-based application, which is designed for MIOS v1.8 or lower, then you will need to migrate the application to support MIOS v1.9

- Extract the 'migration' folder from MIOS source zip file
- Overwrite the files contained in the source of your application.

Take note that this may overwrite customisations you have made to your application, so please take a

backup first, and a copy for comparison with the new files.

ASM

If your application is either:

1. a freshly migrated application (as above)
2. a brand new ASM-based project based on a skeleton \geq v1.9
3. an ASM-based application which already requires MIOS v1.9 or greater (like MBSID v1.7303)

Then the following steps are required:

- Edit mios.h in the source of your application
- Change

```
#define PIC_DERIVATIVE_TYPE 0
```

To

```
#define PIC_DERIVATIVE_TYPE 1
```

- Compile the project  wiki link

- Upload the hex file with MIOS Studio  wiki link

C

If your application is C-based, then the following steps are required. Some are optional recommendations, as noted.

Header and Library

In the case that you should need to take advantage of the additional EEPROM on the newer PICs, the following alterations to the library and header are necessary:

- Edit pic18f452.c in the source of your application
- Change

```
sfr at 0xfa9 EEADR;
```

```
sfr at 0xfab RCSTA;
```

To

```
sfr at 0xfa9 EEADR;
sfr at 0xfaa EEADRH;
sfr at 0xfab RCSTA;
```

- Edit pic18f452.h in the source of your application
- Change

```
extern __sfr __at 0xfa9 EEADR;
extern __sfr __at 0xfab RCSTA;
```

To

```
extern __sfr __at 0xfa9 EEADR;
extern __sfr __at 0xfaa EEADRH;
extern __sfr __at 0xfab RCSTA;
```

Note that the filenames stay as pic18f452., regardless of the PIC model we are actually using. For our purposes, SDCC considers the '4620 to be the same as a '452.*

C-Wrapper

The C-Wrapper will need to be edited as follows:

- In the source of your application, edit mios_wrapper\mios.h
- Change

```
#define PIC_DERIVATIVE_TYPE 0
```

To

```
#define PIC_DERIVATIVE_TYPE 1
```

If you want to use this function, you may want to apply a small fix to the DEC2BCD Helper:

- In the source of your application, edit mios_wrapper\mios_wrapper.asm
- Change

```
global _MIOS_HLP_Dec2BCD

movwf    MIOS_PARAMETER1           //Moves W (the low byte of the 16-bit
integer) into MIOS_PARAMETER1 - That ain't right. See below from the MIOS
Function Reference
movff    FSR0L, FSR2L              //These guys
movf     PREINC2, W                 //Put the high byte in W. D'oh!
```

```
goto    MIOS_HLP_Dec2BCD
```

To

```
global  _MIOS_HLP_Dec2BCD           //The low byte is already in W
movff   FSR0L, FSR2L                //These guys
movff   PREINC2, MIOS_PARAMETER1    //Put the high byte in
MIOS_PARAMETER1. Yay!

goto    MIOS_HLP_Dec2BCD
```

Linker Script

Modifications should be made to the linker script in order to take advantage of the additional capabilities of the 4620/4520. If you are using a standard, PIC18F452-based application, these steps should not be necessary. These procedures are intended for applications being developed which will require the additional capabilities of the newer PICs.

Extend Codepage

Both the 4620 and 4520 have extended code memory. To utilise this fully, make the following alterations:

- In the source of your application, edit project.lkr
- Change

```
CODEPAGE    NAME=page      START=0x3000      END=0x7FFF
```

To

```
CODEPAGE    NAME=page      START=0x3000      END=0xFFFF
```

Add Databanks

In order to give our application the ability to recognise all that lovely, lovely RAM in the newer '4620 and '4520 PICs, one or a mixture of the following options is required:

Standard Bank Size

- In the source of your application, edit project.lkr
- Change

```
DATABANK    NAME=miosram_u  START=0x380      END=0x5FF      PROTECTED
```

ACCESSBANK	NAME=accesssfr	START=0xF80	END=0xFFF	PROTECTED
To				
DATABANK	NAME=miosram_u	START=0x380	END=0x5FF	PROTECTED
DATABANK	NAME=gpr6	START=0x600	END=0x6FF	
DATABANK	NAME=gpr7	START=0x700	END=0x7FF	
DATABANK	NAME=gpr8	START=0x800	END=0x8FF	
DATABANK	NAME=gpr9	START=0x900	END=0x9FF	
DATABANK	NAME=gpr10	START=0xA00	END=0xAFF	
DATABANK	NAME=gpr11	START=0xB00	END=0xBFF	
DATABANK	NAME=gpr12	START=0xC00	END=0xCFF	
DATABANK	NAME=gpr13	START=0xD00	END=0xDFF	
DATABANK	NAME=gpr14	START=0xE00	END=0xEFF	
DATABANK	NAME=gpr15	START=0xF00	END=0xF7F	
ACCESSBANK	NAME=accesssfr	START=0xF80	END=0xFFF	PROTECTED

Extended Bank Capacity

The above change will enable SDCC to allocate the variables in your application to any of the specified banks above. The very observant among you may have noticed that these banks are 256 bits each.... So what happens if you want to use a variable which is greater than 256 bits in size, such as a large array, or string of characters? For this, you will need to create a bank of extended size, and you will need to direct your application to use that bank to store your large variable.

In order to create memory banks of extended capacity, it is necessary to section off a greater range than those given above. A good way to go about this is to combine two or more of the default banks. The following are examples of this.

Making a single, 512-bit bank:

```

DATABANK  NAME=miosram_u  START=0x380      END=0x5FF      PROTECTED
// DATABANK  NAME=gpr6      START=0x600      END=0x6FF
// Remove this bank
// DATABANK  NAME=gpr7      START=0x700      END=0x7FF
// And remove this bank
DATABANK  NAME=gpr67      START=0x600      END=0x7FF
// And create this one out of the two
DATABANK  NAME=gpr8      START=0x800      END=0x8FF
DATABANK  NAME=gpr9      START=0x900      END=0x9FF
DATABANK  NAME=gpr10     START=0xA00      END=0xAFF
DATABANK  NAME=gpr11     START=0xB00      END=0xBFF
DATABANK  NAME=gpr12     START=0xC00      END=0xCFF
DATABANK  NAME=gpr13     START=0xD00      END=0xDFF
DATABANK  NAME=gpr14     START=0xE00      END=0xEFF
DATABANK  NAME=gpr15     START=0xF00      END=0xF7F
    
```

```
ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFF PROTECTED
```

Note that the START of the bank is the same as the START of the first bank removed, and the END of the bank, is the same as the END of the last bank removed.

This can be extended into larger ranges, and multiple customised ranges, as below:

```
DATABANK NAME=miosram_u START=0x380 END=0x5FF PROTECTED
// DATABANK NAME=gpr6 START=0x600 END=0x6FF
// Remove this bank,
// DATABANK NAME=gpr7 START=0x700 END=0x7FF
// And remove this bank,
DATABANK NAME=gpr67 START=0x600 END=0x7FF
// And create this 512-bit bank out of the two 256-bit banks.
DATABANK NAME=gpr8 START=0x800 END=0x8FF
DATABANK NAME=gpr9 START=0x900 END=0x9FF
DATABANK NAME=gpr10 START=0xA00 END=0xAFF
// DATABANK NAME=gpr11 START=0xB00 END=0xBFF
// Remove this bank,
// DATABANK NAME=gpr12 START=0xC00 END=0xCFF
// And remove this bank,
// DATABANK NAME=gpr13 START=0xD00 END=0xDFF
// And remove this bank,
// DATABANK NAME=gpr14 START=0xE00 END=0xEFF
// And remove this bank!
DATABANK NAME=gpr1114 START=0xB00 END=0xEFF
// And create this 1024-bit (1 Kilobit) bank out of the four 256-bit banks.
DATABANK NAME=gpr15 START=0xF00 END=0xF7F

ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFF PROTECTED
```

Or of course you could make the whole lot into one bank if you wanted to:

```
DATABANK NAME=miosram_u START=0x380 END=0x5FF PROTECTED
DATABANK NAME=gpr615 START=0x600 END=0xF7F
// That's almost 2.5kilobits!!

ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFF PROTECTED
```

Add Sections

In order to assist in the use of these memory banks, we can give create 'sections' with names, and those names can be referenced in our code later on. I will use the 2nd example above, to demonstrate:

```

DATABANK    NAME=miosram_u    START=0x380                END=0x5FF                PROTECTED
DATABANK    NAME=gpr67        START=0x600                END=0x7FF
// And create this 512-bit bank out of the two 256-bit banks.
DATABANK    NAME=gpr8         START=0x800                END=0x8FF
DATABANK    NAME=gpr9         START=0x900                END=0x9FF
DATABANK    NAME=gpr10        START=0xA00                END=0xAFF

DATABANK    NAME=gpr1114      START=0xB00                END=0xEFF
// And create this 1024-bit (1 Kilobit) bank out of the four 256-bit banks.
DATABANK    NAME=gpr15        START=0xF00                END=0xF7F

ACCESSBANK  NAME=accesssfr    START=0xF80                END=0xFFF                PROTECTED

SECTION     NAME=CONFIG      ROM=config
// This SECTION entry will already exist in the file. Do NOT alter this
line!

SECTION     NAME=gpr8        RAM=gpr8
// This creates a SECTION called 'gpr8' which references the normal 256-bit
bank 'gpr8'
SECTION     NAME=b512        RAM=gpr67
// This creates a SECTION called 'b512' which references our 512-bit bank
SECTION     NAME=b1024       RAM=gpr1114
// This creates a SECTION called 'b1024' which references our 1kb bank

```

You may create as many or as few sections as you require for your application.

Application Code

Once these sections are created, you can use them within your application, by forcing a variable to be stored within that section. This is done using the 'udata' pragma statement with the following syntax:

```
#pragma udata section_name variable_name
```

For example, referencing the above section:

```

#pragma udata b512 MIDI_Table // This means "store a variable named
'MIDI_Table' in the SECTION named 'b512'
unsigned int MIDI_Table[512]; // Declare the array named 'MIDI_Table',
and now it will be stored in 'b512'

```

Still reading? ;)

From:

<https://wiki.midibox.org/> - **MIDIbox**

Permanent link:

https://wiki.midibox.org/doku.php?id=using_pic18f4620&rev=1152434278

Last update: **2006/10/15 09:35**

