MIGRATING FROM THE PIC16F88

There are many very good articles, tutorials and videos available on the internet that cover the topic of Microchip's 'Enhanced Mid-Range' (EMR) PIC family. The Figures used in this document are screenshots from Microchip Technology's "Introduction to the Enhanced Mid-Range Architecture" PDF, which is old but quite good. (references are specified at end of this document)

But as good as these articles may be, I couldn't find much information detailing how to go about the practical process of migration to such new devices. Microchip actually began the release of this new EMR family about 10 years ago, but like me, I suspect many people had no pressing reason to seriously delve into this new family until now.

Those who are somewhat familiar with mid-range (8 bit) PIC's may find this article useful.

THE VANISHING PIC16F88?

Silicon Chip has had many great and timeless PIC projects over the years, like the L-C Meter, RF Level Meter, a Frequency counter and the Solar PWM chargers but to name a few. Most of these Silicon Chip projects were based on the versatile PIC16f88, as it was inexpensive and had more than enough I/O, ADC and memory to suit the needs of most small projects.

It is well known that Microchip Technology Inc support their microprocessors for many years (if not decades!) after initial release, and as Silicon Chip provided the circuits and source assembler code (*.ASM) for most of these projects, should anything fail I felt confident I could fully maintain these projects for many years to come.

So what could possibly go wrong?

Well, for some reason it seems most local supplies of the PIC16F88 have all but dried up. This may be linked to the general microprocessor chip shortage that the covid19 pandemic has sparked (colloquially known as the 'covid19 supermarket toilet roll" phenomenon), but there are probably lots of other factors, such as the importance of other newer PIC's with better features and lower prices being Microchips first priority.

So, even though you can theoretically still buy 16F88's from Microchip (for about US$4 each), they are also out of stock, and their next production run is scheduled for December 2022! You can still source small quantities via eBay, but with supply so limited, prices are rapidly escalating.

So here lies the problem; the 16f88 is becoming hard to get, and although Microchip have a newer pin-compatible replacement devices available at a very reasonable price (typically only ~US$2 each), the internal architecture has changed to the point that the original Silicon Chip source code is totally incompatible. The 16f88 was based on what Microchip termed their 8-bit "mid-range" (MR) family architecture, but the replacement devices now use the newer "enhanced mid-range" (EMR) architecture.

Now of course many older Silicon Chip 16f88 project designs have been superseded or are no longer viable (for all sorts of reasons), and so the desire to migrate 16f88 code really only arises

if you have a special project that you value …or, if you are like me, you just enjoy tinkering in assembler and want to understand the MR-to-EMR migration process first hand!

So this is where the story begins.

MY BACKGROUND

Pre-retirement, I had a career in RF electronics. I was 'hands on' initially, but much less so in my last couple of decades. The core equipment that I worked on had a growing reliance on microprocessors, either involved in the frequency synthesis, RF channel memory, display generation or in some sort of automation task. So a basic understanding of microprocessors was essential to my work, but the devices we came across were generally pre-programmed industrial devices – and not the sort you would likely find in a Silicon Chip project!

In the early 1980's, I spent many weekends getting to understand how the 6502 and 6800 processors worked at machine code level. At that time, these types of microprocessors were very popular (think of Atari, Commodore 64, Apple II, BBC micro...). However I quickly learned that these early microprocessors were not the ideal platform for simple DIY home projects, as any design required a plethora of associated circuitry, such as data latches, clocks, SRAM, EPROM, I/O, etc.

None the less, this early training in assembler proved to be a good foundation, in that the core knowledge still remains applicable to many current microprocessor platforms. For some reason, I found that higher-level programming languages like Fortran or C somehow felt too abstract and detached, and so despite the passing of the years I have not lost an appreciation of simple and elegant assembly code. To wit, a brilliant example of a useful range-test macro… in just 4 lines of PIC code!

```
Range_chk   macro                   ; Check if pulse is 4500uS +/-10%
            movf    #50uS_COUNT,w   ; W = #count of 50uS edge interrupts
            addlw   255-(4950/50)   ; W= W + (255 - MAX).    [Carry=1 If #count >MAX]
            addlw   (4950/50)-(4050/50)+1 ; W= W+(MAX -MIN)+1. [Carry=0 if #count was<MIN]
            btfss   CARRYBIT        ; (Carry=1 only if within MIN-MAX range)
            endm                    ; at this line, Carry=0 so branch to process a FAIL
                                    ; at this line, Carry=1 so add code to process a PASS
```

A couple of decades later, enter Silicon Chip and several articles by Stan Swan on the "new millenium 555", the new PICAXE-08! The PICAXE-08 was essentially a PIC12f629 (or similar) preloaded with a proprietary bootloader. Programs could be quickly created with a free PIC-basic programmer/compiler, supplied by Revolution Education (RevEd). These PIC's required no need for any extra circuitry apart from perhaps a few switches and LED's. All the flash memory, SRAM and I/O was included in the one chip!

So the PICAXE reignited my interest in microcontrollers, and with my prior familiarity with assembly code programming, I quickly moved to MPLAB assembler programming of 8bit PIC devices like the 16f628 (which even included a UART serial port!) which were fun and easy to manage. My various DIY projects at the time included things like a RS232 sampler/sniffer, a GPS NMEA string decoder, a DDS audio tone generator and the like.

ASSUMPTIONS

This article is aimed to address some of the questions around how to migrate from a 16f88 to a newer extended mid-range device. It is not intended as a tutorial for PIC's, PIC assembler or for the MPLAB IDE. I assume the reader has some knowledge of the 16F88 already, or at least has a basic knowledge of other 8-bit PIC's and of assembly language.

The Microchip documentation website (https://www.microchip.com/doclisting/TechDoc.aspx) provides access to many relevant user guides, and after you install MPLAB IDE on your PC (see https://www.microchip.com/en-us/tools-resources/ for free downloads) you will notice it includes a document directory with lots of useful and relevant documentation.

If you are a glutton for punishment, Microchip also have an online "university" with a wide selection of free courses! (https://mu.microchip.com/page/all-courses)

ENTER MPLAB-X

Around 2015, Microchip upgraded their trusty MPLAB IDE (integrated development platform) programmer to the latest MPLB-X IPE (integrated programming environment) platform.

This is a great boon for C programmers, or those with Mac, Linux and newer Windows operating systems (as the older MPLAB v8.xx was available only in 32bit Windows format)

This new programmer combines support for the entire range of PIC and AVR devices, and includes support for their new range of programmer interfaces and debuggers. Additionally, it has a whole new suite of graphical interfaces, debugging support and lots of C library support.

But as a consequence of all these bells and whistles**, it has become quite bloated with features that users like me will likely never use.** But my biggest disappointment was that there is no support for their MPASM assembly language programmer, nor their trusty old PicKit3 interface! Looking on the Microchip forum sites, it appears I am not alone in lamenting this oversight.

I must point out that MPLAB-X does provide a new XC8 PIC Assembler, which is a free-standing or integrated cross-assembler and linker package. If you are new to assembler, you may love it. But for my part, I am happy to stick with MPASM in MPLAB v8.xx.

Realistically, MPLAB-X is possibly ideal for those modern users with C programming skills and who want to create and debug their code as quickly as possible (after all, time is money!). And this is where the MPLAB-X ties in neatly with the newer PIC EMR devices – many of these EMR architecture changes (like the enhanced indirect addressing capability and virtual linear bank access) are actually designed to allow easier coding under the C language.

But for all of its supposed benefits, C won't necessarily create beautiful, compact, fast & elegant machine code that I am interested in.

BACK TO THE FUTURE – MPLAB IDE

Thankfully, Microchip not only support their older devices, they still provide previous versions of their MPLAB IDE (Integrated Development Environment) programmer for free! See…

https://www.microchip.com/en-us/tools-resources/archives/mplab-ecosystem

Even MPLAB versions of v6.x, v7.x and v8.x are all available at this website, but only the latest versions will have much support for EMR devices.

Keep in mind that the last version of MPLAB IDE was version 8.92 (circa 2014), and it will only recognise and program the devices that had been released at the time. That being said, the good news is it will happily program the popular 12F617, as well as EMR devices like the 16F1459, 16F1509 and the 16F1827/47.

MPLAB IDE v8.92 also supports the PicKit2 or PicKit3 programmers (which you probably have lying around) plus it has some good debugging facilities (like its ICD 3 debugger).

I actually find the most useful debugging tool is the SIM(ulator) tool included in the IDE. Within a couple of key clicks, the simulator runs through your simulated code on screen, while simultaneously showing the status relevant registers or flags in separate windows. You can open windows to view SFR's, clock cycles, stack depth, flags and and variables. Double-click on your source code to add a few breakpoints, and in most cases I find I can prove (or disprove!) my code fairly quickly.

16F88 SUBSTITUTE

All new extended mid-range devices all have the number code of PIC16f**1**xxxx. You may also come across some 8-bit 12f**1**xxxx PIC's too, but these use the 12-bit word flash, not the 14 bit word flash like the 16f88.

Selecting a replacement for the 16f88 turned out to be quite easy. The Microchip website has a parametric comparison tool, it turns out the PIC16f1827 is a good substitute. Eventually I settled on the PIC16f1847 as it has double the flash memory for just a few cents more.

Parameter	16f88	16f1847
DIL Pin Count	18	Same!
Number of I/O Pins	16	Same!
Max freq / CPU MIPS	20MHz / 5MIPS	32MHz / 8MIPS
Data EEPROM (bytes)	256	Same!
SRAM (bytes)	368 in 4 banks	1024 in 32 banks
Program Flash Memory (words)	= 4096 (words) = 2 x 2048 pages	8192 (words) = 4 x 2048 pages
Max ADC Resolution (bits)	10	Same!
ADC Channels	7	12
Capture/Compare/PWM (CCP)	1	2
Number of Comparators	2	Same!

SPI	1	2
Max 16 Bit Digital Timers	1	Same!
Max 8 Bit Digital Timers	2	4
Operating Voltage	2 - 5.5V	1.8 - 5.5V
Microchip price	US$3.42 (but unavailable till Dec2022!)	US$1.95 (in stock)

As for those simpler Silicon Chip projects that used the older 12f675, migration to EMR architecture is not necessary as this chip is freely available and still cheap. However you would be unwise to design a new project based on the 12f675/683, as the newer 12f617 is cheaper and more powerful, so is a far better choice.

SO WHAT IS EMR ARCHITECTURE?

Enhanced mid-range core devices have additional features, additional instructions and generally more flash code space and SRAM. The new EMR architecture is intended to free up some previous architecture limitations and make the devices easier to program in MPLAB-X using the C++ language.

Opcodes

The RISC (reduced instruction set code) architecture was always a pleasant feature of PIC's.

With EMR devices, there are several new opcodes, some are general improvements and some are intended to make C programming easier:

- ADDWFC, SBWFB, LSLF, LSRF, ARSF for Logical/Array/Lookup
- MOVLP, MOVLB provide easier and faster Paging and Banking
- BRA (signed), BRW (unsigned), CALLW provide improved +/- 256 byte relative branching
- ADDFSR0L (signed), MOViW, MOVWi for use with the enhanced FSR0L Indirect Addressing
- RESET is a real opcode now, no more waiting for WDT to timeout & provide a reset

New Instructions

Mnemonic	Description
ADDWFC	Add W+F with Carry
SUBWFB	Subtract F-W with Borrow
LSLF	Logical Shift Left
LSRF	Logical Shift Right
ASRF	Arithmetic Shift Right
MOVLP	Move Literal to PCLATH
MOVLB	Move Literal to BSR
BRA	Branch Relative (signed)
BRW	Branch PC + W (unsigned)
CALLW	Call PCLATH:W
ADDFSR	Add Literal to FSRn (signed)
MOVIW	Move indirect to W
MOVWI	Move W to Indirect
RESET	Reset Hardware & Software

Interrupts

Critical MPU registers are now auto-saved & restored by Interrupt, so there is no need to push/pop these critical registers during an Interrupt Service Routine (ISR). The RETFIE opcode at the end of the ISR now auto-restores ('pops') these critical registers.
These critical registers, as well as the stack and some debug registers are conveniently grouped together on Bank 31.

Common Core Registers

	Address	Register	Function
	0x00	INDF0	Indirect Register 0
NEW	0x01	INDF1	Indirect Register 1
	0x02	PCL	Program Counter Low
	0x03	STATUS	Status Register
NEW	0x04	FSR0 Low	File Select Register 0 Low Byte
NEW	0x05	FSR0 High	File Select Register 0 High Byte
NEW	0x06	FSR1 Low	File Select Register 1 Low Byte
NEW	0x07	FSR1 High	File Select Register 1 High Byte
NEW	0x08	BSR	Bank Select Register
NEW	0x09	WREG	Working Register
	0x0A	PCLATH	Program Counter Latch High
	0x0B	INTCON	Interrupt Control Register

Figure 1: Registers auto-saved and auto-restored on Interrupt

Paging

Mid-range PIC flash memory has always been broken down to 2k 'page' boundaries, as the 14-bit word limitation AND the opcode 'CALL' (3-bits) meant only 11 bits were left for the CALL address. Hence flash memory code must be viewed to be segmented into 2k 'page' boundaries, as this is the address limit of CALLs.

It is recommended to use PAGESEL pseudo-code for all paging, as this makes code fully transportable between MR and EMR devices and allows EMR devices to use the new and faster 'movlp' opcode for paging.

Banking

It is recommended to use BANKSEL pseudo-code for all register selection, as this makes code fully transportable between MR and EMR devices and allows EMR devices to use the faster 'movlb' BSR code for all banking.

PIC's use the Harvard architecture (i.e. separate address space for RAM data and flash program memory), and the RAM and control registers are arranged into 128 byte 'Banks'. With EMR architecture, there are many more Banks in order to accommodate the extra General Purpose RAM (GPR) and the extra control Special Function Registers (SFR's) and still maintain the 128-byte Bank limit.

The multitude of SFR's are now spread over a large range of Banks, but this is not a concern if you use the BANKSEL command when addressing the register, as the assembler will automatically select the correct Bank for you - just remember to revert to Bank 0 when done!

The last Bank (Bank 31) contains critical ISR registers & debug registers available for use.

As usual, the last 16 bytes of SRAM is common to all Banks, but now that idea has been extended so that the first 12 MPU Core Special Function Registers (SFR's) such as the PCL, INDFx, FSRx, Status and W Register are now also common to all 32 Banks.

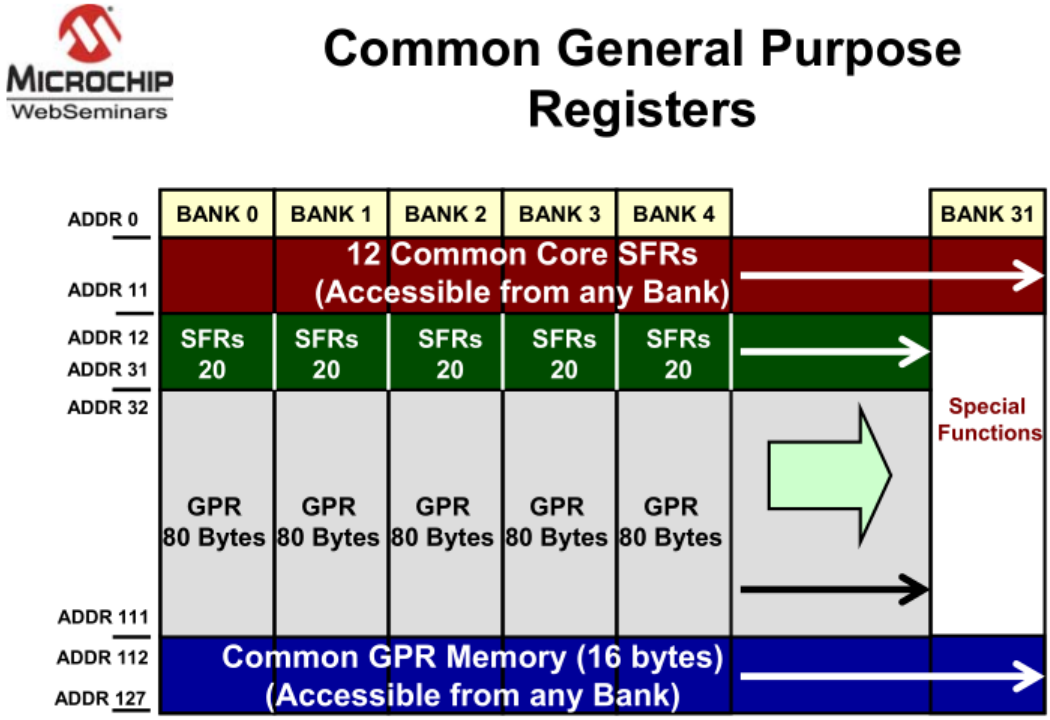

Figure 2: EMR Banks structure with shared SFR's and RAM

Indirect Addressing

With EMR, there are two full 16-bit indirect registers which can access ALL memory, SRAM and flash. (ISR0 points to the 16-bit address in FSRH0/FSRL0, and ISR1 points to the 16-bit address in FSRH1/FSRL1).

These new 16bit FSR registers are located in the "common first 12 bytes of SFR's' (i.e. common to all Banks) so they can be quickly accessed anytime.

The really cool thing about EMR indirect addressing is that the 16 bit FSR registers now allow access to virtually ALL memory of the PIC, SRAM and Flash! It's almost as though the Harvard architecture has been converted into Von Neumann architecture. The EMR FSR register sees ALL memory in the PIC as a single 32k virtual address space, and even the Banked general purpose RAM is mapped into the this space as a contiguous (linear) block of data memory.

Banks appear in the bottom half of the memory map (including new linear mapping of 'virtual' banks), and the flash code memory appear in the top half.

MICROCHIP WebSeminars

(2) 16-bit File Select Registers (FSRs)

Address	Register	Function
0x00	INDF0	Indirect Register 0
0x01	INDF1	Indirect Register 1
0x02	PCL	Program Counter Low
0x03	STATUS	Status Register
0x04	FSR0 Low	File Select Register 0 Low Byte
0x05	FSR0 High	File Select Register 0 High Byte
0x06	FSR1 Low	File Select Register 1 Low Byte
0x07	FSR1 High	File Select Register 1 High Byte
0x08	BSR	Bank Select Register
0x09	WREG	Working Register
0x0A	PCLATH	Program Counter Latch High
0x0B	INTCON	Interrupt Control Register

Figure 3: The two new 16 bit FSR registers and associated INDFx registers

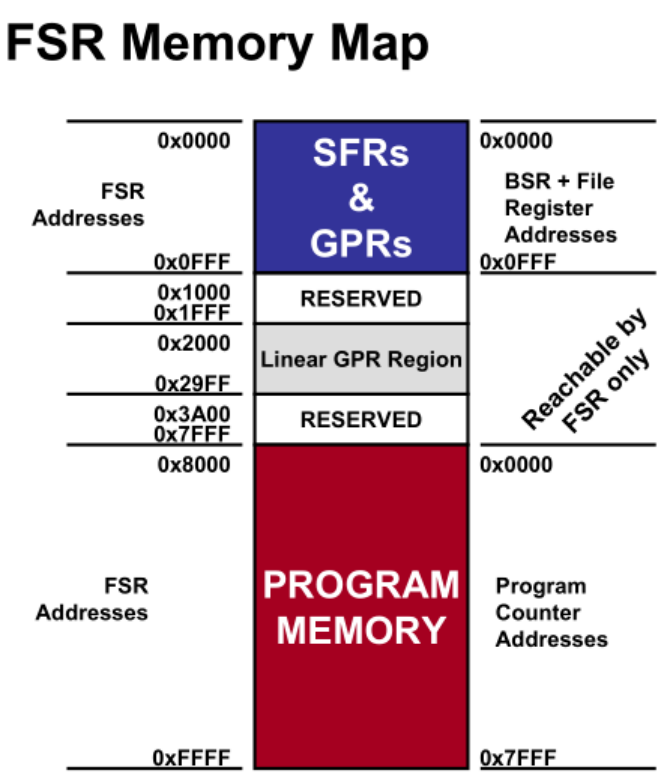

Figure 4: LHS- New FSR 'virtual' address, RHS- Harvard memory addresses

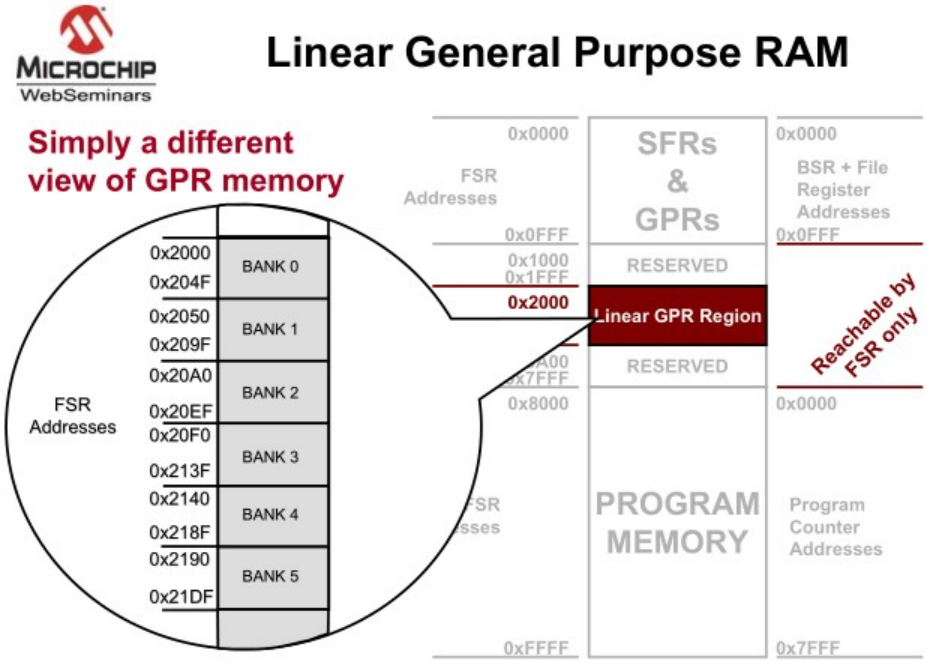

Figure 5: LHS- Linear GPR virtual address space

Note that the legacy STATUS-IRP bit - which was used in the 16f88 for accessing data memory greater than 256bytes (i.e. anything greater than Bank0/1 access) - is no longer present in EMR devices.

With the EMR devices, the fastest way to change your working bank is to manually update the FSR(H) register, however you can also use BANKiSEL (BANKiSEL is transportable from MR to newer EMR devices, but note it doesn't preserve the W register).

LATx Register

A new Latch register added to all I/O ports avoids the previous potential problem of [reading then] writing the INPUT PIN values into the PORT write latches.

With standard MR PICs, it was possible to write a value to a port, then do another bit test (or bit change) to the same port, only to find out that the resultant port state ends up quite unlike what you expected. This is because of the "read-modify-write" problem, whereby the PIC always reads the actual state of the physical pins before modifying and then writing back the new value to the port.

Say you wrote $FF to the port, but some pins were slow to change state (say, because of a large capacitive load), then if you read it back immediately it might only take a microsecond, but that could be too fast for all capacitive loads to charge up to the $FF state you requested. So it was possible for this problem to pop up at unexpected times, even when you thought you were manipulating an unrelated pin on that Port!

So all I/O Ports now have a LATCH register. A write to LATx register instead of the PORTx register eliminates any potential read-modify-write issue. But you can also still read the Port pins directly if you so desire.

ADC

EMR devices often have more AD channels available, although there is still just one central ADC module to do the conversions. There are also new ADC options, like an internal Fixed Voltage Reference (FVR) option & a Temperature sensor.

As a result, the ADC control registers and Analog Select registers (ANSEL) are arranged somewhat differently, but the ADC initialization sequence has not altered.

Weak Pull-Ups

16f88 had an optional WPU only port B (only) when it was configured as an input.

With the 16f1847, each pin on Port B has an individually configurable internal weak pull-up (with global control). Port RA5 also has a WPU option.

Clock

EMR device clocks are often faster, and may have different prescalers or connection methods to the internal timers. So any code that uses the internal clock for Timers, ADC and delays may also need some adjustment.

WDT

The 16f1877 WDT now has a new prescaler for much longer timeouts (up to 256 Seconds!), plus the Config Word now has 4 options for WDT operation, not just two. These are:

```
_WDTE_OFF       EQU  H'3FE7'    ; WDT disabled
_WDTE_SWDTEN    EQU  H'3FEF'    ; WDT controlled by the SWDTEN bit in the WDTCON register
_WDTE_NSLEEP    EQU  H'3FF7'    ; WDT enabled while running and disabled in Sleep
_WDTE_ON        EQU  H'3FFF'    ; WDT enabled
```

MIGRATING TO EMR ARCHITECTURE

So, this is the rough process I used to migrate my 16F88 assembly code to 16F1847 EMR format.

It takes a bit of patience, but if you value your old code the good news it is worth it, and, surprisingly, I converted several thousand lines of 16F88 source code (about 3k words) in just a couple of weekends. Strangely I found the process quite meditative, a bit like doing a moderate Sudoku puzzle…and I solved it!

The biggest scare I had was my initial problem of no display, but that was unrelated to the migrations, it turned out to be (embarrassingly) my incorrect voltage applied to the LCD contrast pin on a new 16x2 LCD.

Because I migrated my older code in logical steps (like suggested below), surprisingly few bugs surfaced during the process.

I used frequently used the Ctrl-H (find and replace) command within MPLAB for most code modification, and this made searching for and replacing redundant legacy code quite easy, despite it having 4000 lines! After each major change, I incremented the ASM filename and recompiled and retested the code in my hardware. In total, only about a dozen or so recompiles were necessary in the end.

Below is the outline and approximate sequence I used for my migration from 16f88 to 16f1847.

Processor Definition & Config Bits

Alter the processor definition and configuration bits (aka 'fuses') at the start of your code..

```
        list P=16f1847
        #include p16f1847.inc

  __CONFIG _CONFIG1, _FOSC_INTOSC & _WDTE_SWDTEN & _PWRTE_ON & _MCLRE_ON & _CP_OFF &
_CPD_OFF & _BOREN_OFF & _CLKOUTEN_OFF & _IESO_OFF & _FCMEN_OFF

  __CONFIG _CONFIG2, _WRT_OFF & _PLLEN_OFF & _STVREN_OFF & _BORV_HI & _LVP_OFF
```

…or you can do it with MPLAB IDE menu after specifying your device [*Configure>Select Device*] then [*Configure > Configuration Bits*].

Watchdog Timer

The WDT now has a new prescaler for much longer timeouts, plus the Config Word now has 4 options for WDT operation. I recommend you disable the WDT until you are satisfied all other code alterations are implemented and are fully working.

CONFIG word declaration for the WDT:

```
_WDTE_OFF          EQU  H'3FE7'     ; WDT disabled
_WDTE_SWDTEN       EQU  H'3FEF'     ; WDT controlled by the SWDTEN bit in the WDTCON register
```

```
_WDTE_NSLEEP        EQU  H'3FF7'     ; WDT enabled while running and disabled in Sleep
_WDTE_ON            EQU  H'3FFF'     ; WDT enabled
```

Main code initialization of WDT:

```
    banksel   WDTCON
    movlw     b'00010110'       ; S/W WDT (31.5khz INTRC clk/65556) = 2.09S timeout
    movwf     WDTCON            ; WDT prescaler (WDT enabled in config word)
    bsf       WDTCON,0          ; SW enable WDT
```

Clock

The 16f1847 can now run as high as 32MHz, but I recommend you stick to the old 16f88 speed until safe. Any code that relies on the internal clock (eg Timers, ADC and delays) may also be impacted and may need adjustment.

Refer to the 16f1847 datasheet to initialise the clock registers.

```
    banksel  OSCCON          ; Bank1
    movlw    b'01110010'     ; 8MHz, running from INTRC
    movwf    OSCCON          ; set clock
    movlw    b'00000000'
    movwf    OSCTUNE
```

Ports, LATx and WPU's

With the new LATx latch register is present on all I/O ports, I recommend you read and write to LATx (and not directly to the Port) unless you are sure it's safe from the read-modify-write issue described earlier.

The initialisation of the Ports and the TRIS (tristate, or data direction) register are unchanged.

With the 16f1847, each pin on Port B now has an individually configurable internal weak pull-up, as does Port RA5. Take this into consideration when looking at your hardware design.

```
; set portA & B data direction & pullups
    banksel   LATA              ;
    clrf      LATA              ;Data Latch PORTA
    clrf      LATB              ;Data Latch PORTB

    banksel   WPUA
    clrf      WPUA              ; disable all WPU (not needed on A/D channels)
    clrf      WPUB              ; disable all WPU (not needed on Act-Hi SW input RB2)
; set port data direction, where (0)=outputs and (1)=inputs
    banksel   TRISA             ; Bank1 - DataDir register
    movlw     b'00111111'       ; <7-6> outputs, all else inputs
    movwf     TRISA
    movlw     b'00000100'       ; SW_IN<2>, all else outputs
    movwf     TRISB
```

ADC

EMR device ADC's have new options, like an internal Fixed Voltage Reference (FVR) option & a Temperature sensor. The FVR is a welcome improvement to consider instead of using Vdd as the +ve ADC voltage reference. For example, if your old hardware design used a78L05 as the ADC +ve reference, you could have a +/-4% error on all ADC readings (the 78L05 was typically 5V +/-0.2V at 25C!).

As there are five extra ADC channels on the 16f1847 (12 in total) and the ADC control registers are a bit different, there are now two analog initialization registers (ANSELA/B) and two ADC control registers (ADCCON0/1) to initialise. For example…

```
; ....................................................
        banksel  FVRCON          ;  setup FVR as an optional Vdd +ve reference
        movlw    b'10000011'     ; set FVR ENabled & ADC buffer gain x4
        movwf    FVRCON
; ....................................................

        banksel  ANSELA          ; Bank3
        movlw    b'00011111'     ; setup PortA <4:0> as analog inputs
        movwf    ANSELA
        clrf     ANSELB          ; (portB currently not used for any analogue)

        banksel  ADCON0          ; Bank1
        movlw    b'00000000'     ; default is ChanSel 000 (AN0) & set ADC to off
        movwf    ADCON0
        movlw    b'11010000'     ; right justified result,(Fosc/16) = 2uS AD clk,  AD Ref = Vss-Vdd
;       movlw    b'11010011'     ; as above, but with option for AD Ref = Vss-Vfvr (4.096v)
        movwf    ADCON1
        bsf      ADCON0,ADON     ; A/D converter enabled!
```

Banks and Page references

Remove any Bank0/1/2/3 DEFinitions (or Bankx references) & replace with BANKSEL [registername] statements.

Remove any Page0/1 DEFinitions (or Pagex references) & replace with PAGESEL [label] statements.

Initialisation of misc. CONFIG registers

Depending on your application, you may have some various timers or interrupt registers to setup. Just compare the old and new datasheets to check if anything has changed.

And if your original source code specifies the setup of explicit bits in config registers, it's more future-proof if you recode those to use the generic Microchip labels instead, as the assembler will substitute the correct bit-field should you swap devices.

```
;This method of coding may work on your old processor....
                bank0
                bsf      ADCON0,0                ; A/D converter enabled
```

```
;…but this is a more transportable way to write the same code….
                banksel  ADCON0
                bsf      ADCON0, ADCON          ; A/D converter enabled
```

Macros

Check Macros for any old Bank or Page switching code, and ensure it only uses Banksel and Pagesel commands as described above.
You should also consider creating new Macros if the new frequent use of Banksel [register] command makes your main code messy and harder to read.

For example, one piece of my main code spent a lot of time altering a PWM register. Creating new macro definitions of read_PWM, write_PWM , inc_PWM etc meant I still did all the necessary new Banksel swaps, but my new main code stayed clean and easier to read.

```
read_PWM    macro                   ; Example in removing messy bank swaps
            banksel   CCPR1L        ; Bank5
            movf      CCPR1L,w      ; load W from PWM
            banksel   d'0'          ; Bank0
            endm
```

Interrupt Service Routines (ISR's)

Critical registers are now auto-saved in BANK32 and are auto-restored on RETFIE, so ISR's no longer need any code to push/pop the critical registers.

But if you are concerned, leaving the old push / pop code as it was won't hurt anything either.

Keep in mind that deleting push/pop code not only speeds up the ISR code, it can also mean that by deleting any old ISR RAM variables in the valuable $70-$7f shared RAM space gives you the option of reusing that valuable space. (ie you can repurpose that RAM space for actions that would otherwise require a lot of Bank switching to manipulate).

Indirect addressing considerations

With the replacement of the old INDF register with the new 16-bit INDFx registers, it opens all sorts of possibilities for code improvement. In my case, this gave me the opportunity to simplify my LCD string display code and place my LCD strings lookup table on any page I wanted. Plus I could also remove the legacy RETLW code (as was often used in past lookup tables) allowing the table to only comprise clean strings of text.

```
; ************************************
STRINGOUT:              ; send string to LCD
; ************************************
; This code intended for extended mid-range devices. The string table can be placed on any page!
```

```
; An ORG declaration & STR_TABLE: label must be placed at start of string table (16 x 16 max)
; Enter with W = OFFSET from TABLE_BASE (the start of the required 16chr string block)
; NOTE each string must start at $xx00 and be 16characters long (or be terminated by $00)

        movwf   FSR1L           ; save LO byte OFFSET for table read
;       movlw   0x0f            ; (optional way to set HI byte for table @ $0F00)
        movlw   HIGH STR_TABLE  ; get HI byte of table start address
        movwf   FSR1H           ; save as HI byte for table read

STROUT_LOOP
        movf    INDF1,w         ; get a character
        btfsc   ZEROBIT         ; is it a string terminator ($00)?
        goto    STROUT_END      ; yes, so exit early
        call    LCD_CHR         ; else display it
        incf    FSR1L           ; inc string pointer
        movf    FSR1L,w         ; get updated string ptr
        andlw   b'00001111'     ; chk LO nibble, has 16 char limit rolled over?
        btfss   ZEROBIT         ; if ZERO, then all done, exit
        goto    STROUT_LOOP     ; else loop & get next character

STROUT_END
        return


; *************************************
; STRING LOOKUP TABLE
; *************************************
STR_TABLE:
                org     0x0f00
STRING_00:      dt    "This is a string"        ; string 1
STRING_10:      dt    "of 16 characters"        ; string 2
;...and so on up to 16 text strings

STRING_E0:      dt    "This 2nd last  "         ; string 15
STRING_F0:      dt    "This is the last"        ; string 16
STR_TABLE_END:
; *************************************
```

REFERENCES:

Microchip: Enhanced Mid-Range PIC MCU Architectural Overview
https://microchipdeveloper.com/8bit:emr-architecture

Microchip WebSeminar: Introducing the Enhanced Mid-Range Architecture
https://www.microchip.com/stellent/groups/sitecomm_sg/documents/devicedoc/en542713.pdf

Microchip PIC1xF1xxxx Software Migration
Microchip Archive, Document Number: DS41375A.pdf