
MPASM Assembler

**with MPLINK and MPLIB
Linker and Librarian**

What is <A> ?

- **Assembly Language**
 - Instructions for a μ P written in the form of mnemonics
 - Confusingly also referred to as “assembler”, as in “assembler code”, or to “... program in assembler ...”
- **Assembler**
 - A program that translates from an assembly language to machine instructions
 - A *Cross Assembler* is a program that runs on one type of processor (e.g. x86) and produces machine instructions for another type (PIC)
- **Assemble**
 - Translate to machine instructions (an assembly language is assembled, a HLL is compiled or interpreted)
- **Assembly**
 - The process of translation

What is an Assembler?

- At least: a translator from *mnemonics* to binary instructions

ADLW h'AA' \Rightarrow 00001111 10101010

- Invariably, an assembler:
 - Has a set of *directives* that control assembler processing
 - Calculates relative addresses from instruction labels
- Most assemblers are *macro assemblers*
 - Perform macro *substitution, expansion* and *calculation* at **assembly time**
 - Macro language allows assembly language programming at a higher level of abstraction

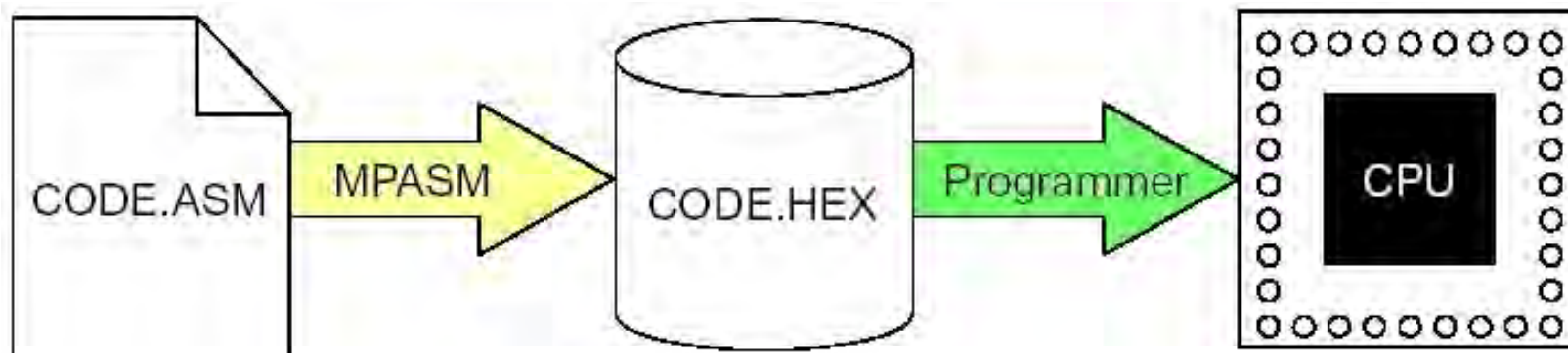
```
local i = 0           ; establish local index variable and initialize
while i < 8           ; do <something> 8 times
    <something>
    i += 1            ; increment loop counter
endw                  ; break after eight loops
```

- *Structured assembler* – see Peatman for example and source code
 - Very simple form of compiler
 - Allows control structures (e.g. if-then-else) that are active at **run time**

Simple Workflow - Assembler

With a single assembly language file that references only *absolute* addresses, workflow is straightforward

- Assemble to absolute object file using MPASM
- Load to device – we will use the MPLAB IDE



MPASM Assembler Files

- **.asm** Assembly language source

Output from Assembler:

- **.lst** Assembler listing file
- **.err** Assembler error messages
- **.o** Relocatable object file

(Output from Linker)

- **.hex** Absolute machine code, Intel Hex format
- **.cof** Absolute machine code, in Common Object File Format – contains executable code and symbol table
- **.map** Load map file – shows where program and data objects are placed in memory

Assembler Listing File (.lst) Format

MPASM 03.20.07 Released

MANUAL.ASM

3-23-2003

18:33:38

PAGE 1

LOC	OBJECT	CODE	LINE	SOURCE	TEXT
-----	--------	------	------	--------	------

VALUE

			00001	; Sample MPASM Source Code.	
			00002	;	
			00003	list	p=18F452
0000000B			00004	Dest	equ 0x0B
000000			00005	org	0x0000
			00006		
000000			00007	Start:	
000000 0E0A			00008	movlw	0x0A
000002 6E0B			00009	movwf	Dest
000004 EF00 F000			00010	goto	Start
			00011		
0001FE			00012	org	0x01FE
0001FE EF00 F000			00013	goto	Start
			00014		
			00015	end	

Assembler Listing File (.lst) Format

MPASM 03.20.07 Released

MANUAL.ASM

3-23-2003

18:33:38

PAGE 2

SYMBOL TABLE

LABEL	VALUE
Dest	0000000B
Start	00000000
__18F452	00000001

MEMORY USAGE MAP ('X' = Used, '-' = Unused)

0000	:	XXXXXXXXX	-----	-----	-----	-----
01C0	:	-----	-----	-----	-----	-----XX
0200	:	XX	-----	-----	-----	-----

All other memory blocks unused.

Program Memory Bytes Used: 12

Program Memory Bytes Free: 32756

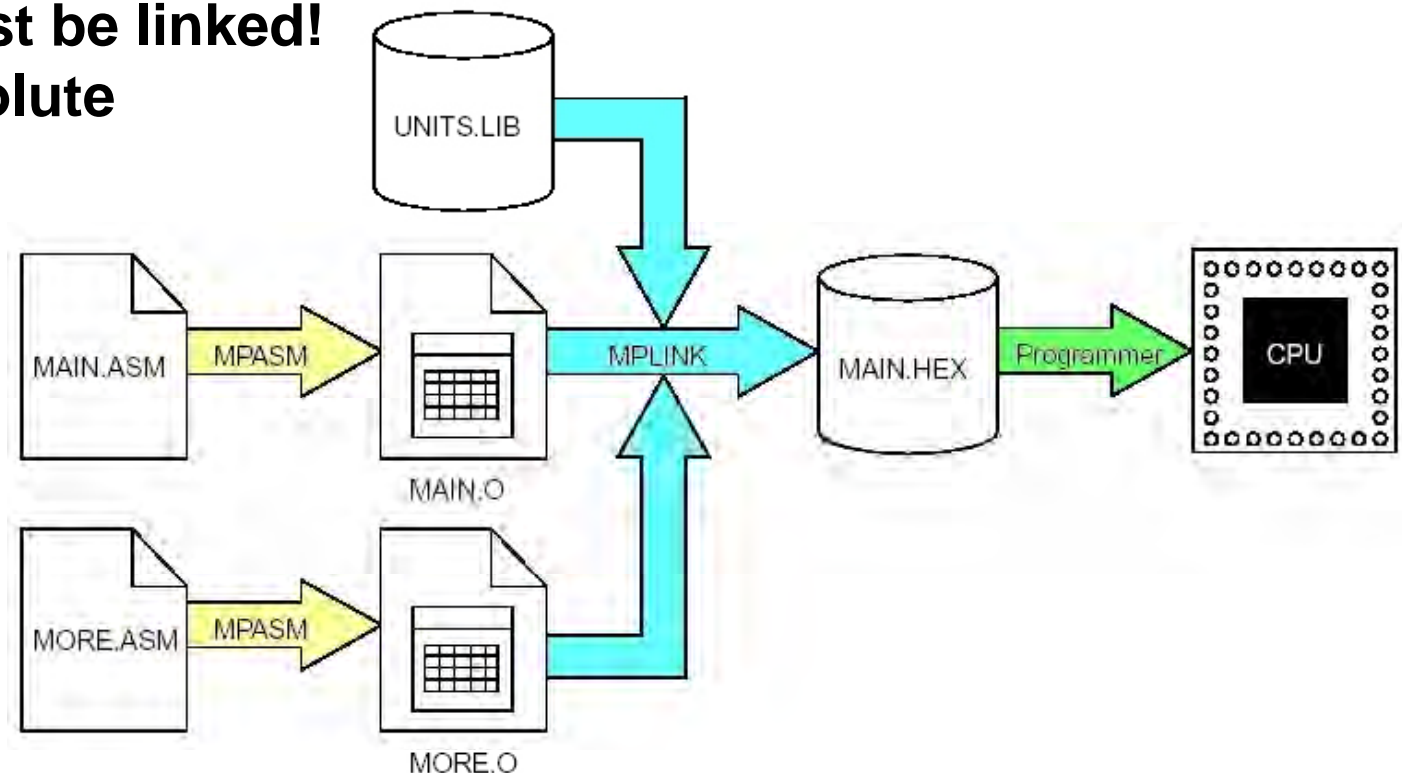
Errors	:	0		
Warnings	:	0 reported,	0 suppressed	
Messages	:	0 reported,	0 suppressed	

What is <L> ?

- **Linker**
 - Program that translates one or more relocatable object modules into executable instructions with absolute addresses
- **Library**
 - Collection of relocatable object modules
- **Librarian**
 - Program that creates and manages a library
 - Add, remove, replace, list object modules

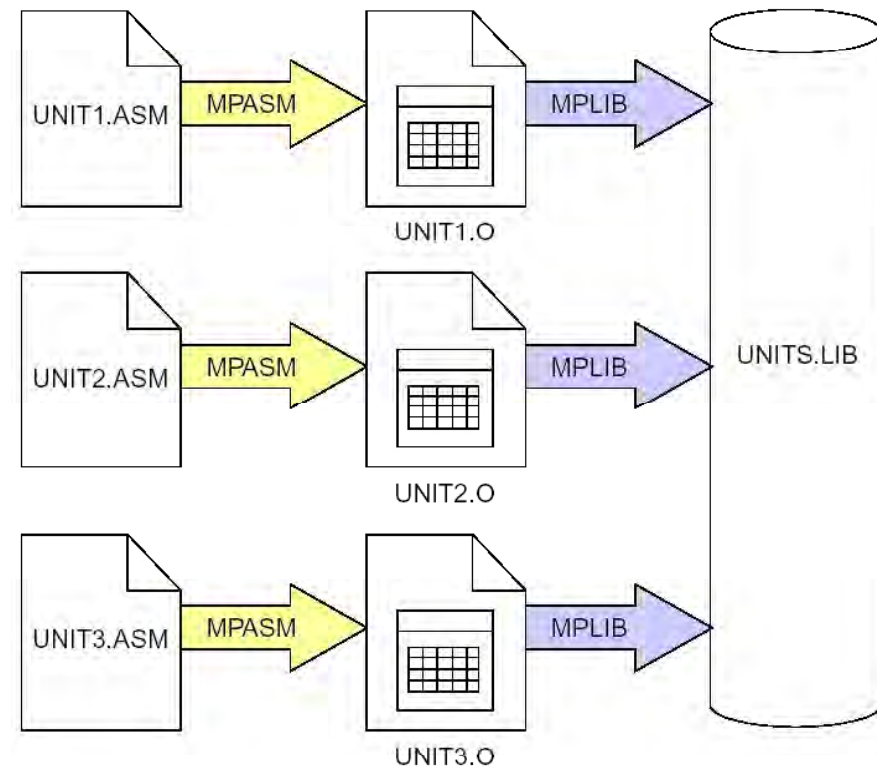
Workflow – Assembler and Linker

- Assemble one or more assembly language Source Files xxxx.asm to **Relocatable** Object Files using MPASM
- Link relocatable object files with linker MPLINK to form absolute executable file
- Even **one file** must be linked! (to calculate absolute addresses)



Workflow – Librarian

- Librarian MPLIB can be used to form libraries of relocatable object code.
- For example:
 - `clib.lib`
 - `p18f452.lib`
 - `myLib.lib`



The Assembler: MPASM

- **Universal macro assembler for *all* PIC devices**
 - Device capabilities and mnemonics (of course) change from one device to another...
- **Choice of three interfaces**
 - Command-line (DOS shell) interface
 - Stand-alone MS-Windows application
 - Integrated with Microchip's MPLAB IDE
- **Free (part of MPLAB IDE)**

MPLAB X Help Files

- **MPLAB X is Microchip's (free) IDE**
- **Available from UoS website**

- **The on-line help files in MPLAB X are good**
- **Go to Help ⇒ Contents ⇒ MPASM Toolsuite**

Assembler Syntax

Rules for assembly language source code

Syntax: Assembly Language File

- Each line of code may contain zero or more
 - Labels
 - Mnemonics
 - Operands
 - Comments
- Maximum line length is 255 characters
- Whitespace is not significant

Wait:

```
    btfss    PIR1, ADIF        ; wait for A/D conversion done
```

Syntax: Assembly Language File

- **Labels**
 - Must start in column 1
 - Are case-sensitive by default
 - Must begin with alphabetic character or underscore (_)
 - Can be 32 characters long
 - Can be followed by a colon (:) or whitespace
- **Mnemonics**
 - Must *not* start in column 1
 - Must be separated from label(s) by colon (:) or whitespace
- **Operands**
 - Must be separated from mnemonics by whitespace
 - Multiple operands must be separated by a comma (,)
- **Comments**
 - Can start anywhere
 - Everything from a semicolon (;) to the end-of-line is a comment

Syntax: Assembly Language File

Radixes (Bases)

- Hexadecimal: H' A3 ' or 0xA3
 - Decimal D' 163 '
 - Octal: O' 243 '
 - Binary: B' 10100011 '
- Note – all the same number,
but different radixes!
- Default radix is Hexadecimal
 - There is **no floating point** type
 - ASCII Character: 'C' or A'C'
 - ASCII String: "A String"
 - Note! **A string is not Null-terminated** unless you add the terminator!

Assembler Control – Radix

RADIX: Specify Default Radix (Base)

Radix <default_radix>

- options are **hex**, **dec**, **oct**
- if not specified, radix defaults to **hex**

Assembler Directives

Directives =
Instructions in the source code that tell the
assembler *how* to assemble a source file

Assembler Directives

- Assembler directives are placed in the assembly language source file, and tell the assembler how to assemble the source.
- They are **not active at run time**
- There are five types of assembler directives, for:
 - Assembler Control
 - Conditional Assembly
 - Data Definition
 - Macro Definition
 - Object File Control

Assembler Control – Symbols

EQU: Defines a Symbolic Label for a Constant

`<label> equ <expr>`

- `expr` is a **number**
- See examples in `P18452.INC`

```
HEIGHT    equ    D'17'  
DEPTH     equ    HEIGHT * 2
```

SET: Defines a Symbolic Label for a Variable

`<label> set <expr>`

- Same as `EQU` except that value of `<label>` can be redefined with another `SET`

```
Length    set    2  
Area      set    HEIGHT * Length  
Length    set    Length + 1
```

Assembler Control – Symbols

CONSTANT: Declare Symbol Constant

```
constant <label> = <expr>  
    [, <label> = <expr>]
```

- A constant must be initialised when defined, and cannot be changed

```
constant BuffLen = D'512'
```

VARIABLE: Declare Symbol Variable

```
variable <label> [= <expr>]  
    [, <label> [= <expr>]]
```

- A variable does not need to be initialised when defined (as in SET), and value can be changed subsequently
- Variable value must be formed before being used as an operand

```
variable RecLen = D'64'  
variable Memory = RecLen * BuffLen
```

Assembler Control – Placing Code

ORG: Set Absolute Program Origin

`<label> org <expr>`

- Sets the value of the assembler's *location counter*
- The location counter value at assembly time corresponds to the PC value at run time
- Cannot be used when generating a relocatable object file
- Use CODE, UDATA, IDATA directives instead

```
org      0x000008
goto HighISR
; HighISR replaced by address
```

```
HighISR:
; High priority ISR goes here
...
RETFIE
```

Assembler Control – Defines

#DEFINE: Define a **Text Substitution Symbol**

#UNDEFINE: Delete a Text Substitution Symbol

```
#define <symbol> [<string>]
```

```
#undefine <symbol>
```

- Same mechanism as in ANSI C
- <string> will be substituted for <name> from the point where #defined

#IFDEF: Execute if Symbol is Defined

#IFNDEF: Execute if Symbol is Not Defined

#ENDIF: Terminates Conditional Block

```
ifdef <symbol>
    <something>
endif
```

```
#define    MAX_INT    D'65535'
```

```
#define    DEBUG
```

```
#ifdef DEBUG
```

```
    constant BuffLen = 8
```

```
    variable RecLen  = 4
```

```
#else
```

```
    constant BuffLen = D'512'
```

```
    variable RecLen  = D'64'
```

```
#endif
```

```
#undefine DEBUG
```

Assembler Control – Include

INCLUDE: Literally include a file at this point

```
#include <path\filename>
```

```
#include "path\file"
```

```
#include path\file
```

- Similar to ANSI C
- No difference in behaviour between the forms <file> and "file" and file
- Search path is current directory; source file directory; MPASM executable directory

```
; get register symbols, etc
```

```
#include P18F452.INC
```

```
; get config bits
```

```
#include config.inc
```


Assembler Control – Listing File

TITLE: Specify Program Title for listing

SUBTITLE: Specify Program Subtitle for listing

`title "<title_text>"`

`subtitle "<subtitle_text>"`

- If defined, `title` and `subtitle` print on each page of the listing

SPACE: Insert Blank Lines

PAGE: Insert Page Eject

`space <expr>`

`page`

- `space` inserts a number of blank lines into the listing file
- `page` inserts a new page character into the listing file

```
title "Code Release 2006-03-16"  
; Stuff here
```

```
page  
subtitle "Memory Diagnostics"  
; Memory Diagnostic code here
```

Assembler Control – Listing File

LIST: Turn on listing, with options

NOLIST: Turn off listing

```
list [<option1>[, <option2>  
      [, ...] ] ]
```

`nolist`

- `list`, with no options, turns listing on
- Options (13 of them) control various listing settings – see Data Sheet
- `nolist` turns off listing

EXPAND: Expand Macros in Listing

NOEXPAND: Don't Expand Macros in Listing

```
expand or noexpand
```

- Expand or suppress expansion of all macros in listing file

```
; suppress listing of symbols, etc  
nolist  
include P18F452.INC  
list
```

Assembler Control – Listing File

MESSG: Create User-defined Message

ERROR: Issue a User-defined Error Message

`messg "message_text"`

`error "error_text"`

- Both print user-defined messages

```
if size > MAX_INT
    error "16-bit value exceeded"
endif
```

Assembler Control – Configuration

RADIX: Specify Default Radix (Base)

`Radix <default radix>`

- options are `hex`, `dec`, `oct`
- if not specified, radix defaults to `hex`

ERRORLEVEL: Set Diagnostic Message Level

`errorlevel 0 | 1 | 2 | <+ -> <msg_number>`

- 0 is show all errors and warnings, 2 is show none
- **Important hint!!** Use `errorlevel 0`
- `-<msg_number>` suppresses a single message

The low, high and upper Operators

LOW: Return the low byte (bits <7:0>)
of a multi-byte value

HIGH: Return the high byte (bits <15:8>)
of a multi-byte value

UPPER: Return the upper byte (bits <21:16>)
of a multi-byte value

table:

```
data "I'm a"  
data "beatles"  
data "eater"
```

```
movlw UPPER table  
movwf TBLPTRU  
movlw HIGH table  
movwf TBLPTRH  
movlw LOW table  
movwf TBLPTRL
```

The banksel Directive

BANKSEL: Generate bank selecting code (`movlb`) that selects the correct bank for a variable in any bank of RAM

```
banksel label
```

```
banksel Var1 ;Select correct bank for Var1  
movwf Var1   ;Write to Var1
```

Assembler Control – Termination

END: End of Assembly Language Program

`end`

- Every program must finish with an `end` directive
- Everything after `end` is ignored

Conditional Assembly Directives

- **Permit sections of code to be conditionally assembled**
- **Similar to C language – e.g.**

`#ifdef`

`...`

`#endif`

Conditional Assembly – If-Else

Have already seen IFDEF, IFNDEF

IF: Begin Conditionally Assembled Block

ELSE: Begin Alternative Block to IF

ENDIF: End Conditional Assembly Block

```
#if <expr>
    <assembly_code>
[#else
    <alternative_assembly_code> ]
#endif
```

- <expr> is evaluated – non-zero is interpreted as logically TRUE

Conditional Assembly – While

WHILE: Loop While <expr> is TRUE

ENDW: End of a WHILE Loop

```
while <expr>
    <assembly_code>
endw
```

- <expr> is evaluated – non-zero value is interpreted as logically TRUE
- <assembly_code> cannot exceed 100 lines
- Can not loop more than 256 times
- Active at **assembly time**

The Five ENDS

- **Don't confuse the various ENDx directives!!**

END:	End of the Assembly Program	any program \Rightarrow end
ENDIF:	End of a Conditional Block	if \Leftrightarrow endif
ENDW:	End of a While Loop	while \Leftrightarrow endw
ENDM:	End of a Macro Definition	macro \Leftrightarrow endm
ENDC:	End an Automatic Constant Block	cblock \Leftrightarrow endc
	(a cblock is used to define a list of named constants)	

Data Definition Directives

- **Control memory allocation and symbol definition**
- **This is how to define named “variables”**

Data Definition - Integers

DB: Declare a Byte of Data

DW: Declare a Word of Data

DE: Declare a Byte of EEPROM Data

`db <expr> [,<expr> , . . . , <expr>]`

`dw <expr> [,<expr> , . . . , <expr>]`

`de <expr> [,<expr> , . . . , <expr>]`

- all **reserve storage** in program or data memory and **initialize the memory** location(s)
 - `db` packs 8-bit values into 16-bit memory.
 - `dw` behaves like `db` for PIC18 devices
 - `dw` packs words into data memory in low-byte/high-byte order
 - See the `idata` directive
 - `de` places 8-bit values into EEPROM

```
; Absolute code in FLASH
```

```
org 0x2000
```

```
errorFlags: db B'10100011'
```

```
highLimit: dw D'350'
```

```
aString: db 'Hello Room!'
```

```
; Relocatable code in RAM
```

```
udata_acs
```

```
accessVar: db 0x55
```

```
udata 0x300
```

```
myVariable: db D'99'
```

Data Definition – Strings, Tables

DA: Store Strings in Program Memory

```
da <expr> [,<expr2>,...,<exprn>]
```

- generates packed 14-bit numbers representing **7-bit** ASCII characters

DT: Define Table

```
dt <expr> [,<expr>,...,<expr>]
```

- `dt` generates a series of `RETLW` instructions, one for each `<expr>`, which must be an 8-bit value. When each is executed, returns with the value in `WREG`.

```
aString:
```

```
    da    "rubric"
```

```
aNullTerminatedString:
```

```
    da    "Null terminated", 0
```

```
squares:
```

```
    dt    0, 1, 4, 9, 16, 25, 36
```

Data Definition – General

DATA: Create Numeric and Text Data

```
data <expr>,[,<expr>,...,<expr>]
```

```
data "<text_string>"  
    [,"<text_string>","..."]
```

- General data definition - places numeric or text data into *Program Memory*
- Single characters placed in low byte of word
- Strings packed 2 characters per 16-bit word, first character in LSB
- Can be used to declare values in `IDATA`

FILL: Fill memory block with value

```
fill <expr>, count
```

- If bracketed by parentheses, <expr> can be a (16-bit long) assembly language instruction

```
data    'C'           ; one character
```

```
data    "Sharp"       ; string
```

Numbers:

```
data    1, 2, 3       ; some numbers
```

```
fill    0x5555, D'10'
```

```
fill    (goto 0), NEXT_BLOCK-$
```

Data Definition – Un-initialised Memory

RES: Reserve Memory

```
res <mem units>
```

- Reserve a number of bytes of memory
- Do not initialize the memory
- In absolute code, Program Memory will be reserved
- In **relocatable** code, memory can be either in Program Memory or Data Memory
- See `code` directive (Program Memory) and `udata` directive (Data Memory)

```
; Absolute code
```

```
org 0x2000
```

```
res 0x20 ; 32 bytes
```

```
; Relocatable code
```

```
Globals:
```

```
udata
```

```
temp res 1
```

```
time res 2 ; 2 bytes
```


Data Definition – μ C Configuration

PROCESSOR: Set Processor Type

`processor <processor type>`

CONFIG: Set Processor Configuration Bits

`config <bit>=<value>`

`__config` is deprecated

- Sets the configuration bits
- Processor must previously have been declared
- If done in the code, MPLAB IDE settings are over-ridden – see `config.inc` on server
- See usage, definitions in `18F452.INC`

`__IDLOCS`: Set values of processor ID Locations

- Similar to `CONFIG`

Data Definition – RAM Configuration

__MAXRAM: Specify maximum RAM address

`maxram <expr>`

- Specifies the highest address of physical RAM

__BADRAM: Specify invalid RAM addresses

`__badram <expr>`

- Can have more than one `__badram` directive
- `__maxram` and `__badram` together allow strict RAM address checking

```
processor    18F452
__MAXRAM    H'FFF'

; Unimplemented banks
__BADRAM    H'600'-H'F7F'

; Unimplemented SFRs
__BADRAM    H'F85'-H'F88'
__BADRAM    H'F8E'-H'F91'
__BADRAM    H'F97'-H'F9C'
__BADRAM    H'FA3'-H'FA5'
__BADRAM    H'FAA'
__BADRAM    H'FB4'-H'FB9'
```

Macro Definition Directives

- **Control execution and data allocation within macros**
- **Description of the macro language follows this section.**

Macro Definitions

MACRO: Declare a Macro Definition

ENDM: End a Macro Definition

```
<label> macro [<arg>,...,<arg>]  
    <statements>  
endm
```

LOCAL: Declare Local Macro Variable

```
local <label> [,<label>]
```

- Declared inside a macro – local scope

EXITM: Exit from a Macro

```
exitm
```

- Forces immediate exit from macro during assembly

```
len      equ 10          ; global  
size     equ 20  
  
m_buffer:  
    macro    size  
        local len, label  
len      set  size      ; local len  
label    res  len  
len      set  len - size  
endm
```

Macro Language

**A simple form of preprocessor / compiler that
allows a limited higher-level abstraction**

Macros

- **Allow “functions” with “arguments”**
- **Macro processor can function like a simple compiler**
- **In reality, macro processor is just doing substitutions – “macro expansion”**

Macro Syntax

- **Syntax is**

```
<label> macro [<arg1>, <arg2>, ..., <argn>]  
    <statements>  
endm                ; ends macro definition
```

- **<label> is the name of the macro**
- **Zero or more arguments**
- **Values assigned to arguments when macro is invoked are substituted for the argument names in the macro body**
- **Body <statements> may contain**
 - **Assembly language mnemonics**
 - **Assembler directives**
 - **Macro directives (macro, local, exitm, endm – only legal in macro)**

Macro Example – Definition

- **Macro definition is**

```
#include "18F452.INC"

;
; Compare register aReg to a constant aConst and
; jump to aDest if register value >= constant.
;

mCmpJge:      macro aReg, aConst, aDest
               movlw      aConst
               subwf      aReg, w
               btfsc      status, carry
               goto       aDest
               endm
```


Macro Example – Invocation

- When invoked (“called”) by:

```
mCmpJge switchVal, maxSwitch, switchOn
```

it will produce (expand to):

```
movlw    maxSwitch
subwf    switchVal, w
btfsc    status, carry
goto     switchOn
```

Relocatable Object Files

Relocatable Object Files

- **Assembled (or compiled) object files with no associated absolute load addresses**
- **Required for**
 - **Building pre-assembled object libraries with MPLIB**
 - **Linking assembly language and C language modules – Compiler output will be relocatable**
- **Specify the *segment* (or *section*) for placement of each part of the linker output, rather than absolute addresses**

Relocatable Object Files: Assembler Directives

So we need directives to work with

- **Projects with multiple assembly language files**
- **Placing information into Program or Data Memory**
- **Relocatable Object Files**

Here they are...

Directives for Object File Imports/Exports

EXTERN: Declare an Externally Defined Label

```
extern <label> [, <label> . ..]
```

- Use when generating relocatable object file
- Similar to C/C++ extern – declare a label (name of subroutine, etc) that is declared outside the file being assembled
- Resolved by the linker

GLOBAL: Export a Label to Linker

```
global <label> [, <label>]
```

- Use when generating relocatable object file
- Declare a label (name of subroutine, etc) to make it visible outside the file being assembled
- Resolved by the linker

Directives for Object File Memory Segments

CODE: Begin an Executable Code Segment

`[<label>] code [<ROM_addr>]`

- If <label> unspecified, defaults to `.code`
- Starting address initialised to <ROM_addr>, or at link time if no address specified

UDATA: Begin Un-initialised Data Segment

`[<label>] udata [<RAM_addr>]`

- If <label> unspecified, defaults to `.udata`
- Declares a segment of Un-initialised data
- Starting address initialised to <RAM_addr>, or at link time if no address specified

```
; Relocatable code - variable
; in RAM
        udata
aVariable:  res  1
```

Directives for Object File Memory Segments

UDATA_ACS: Begin Object File Un-initialised Data
Segment in Access RAM

`[<label>] udata_acs [<RAM_addr>]`

- If <label> unspecified, defaults to `.udata_acs`
- Declares a segment of Un-initialised data in *Access RAM*
- Starting address initialised to <RAM_addr>, or at link time if no address specified

```
; Relocatable code - variable
; in access RAM
        udata_acs
accessVar: res 1
```

UDATA_OVR: Begin Object File Un-initialised Data
Overlay Segment

`[<label>] udata_ovr [<RAM_addr>]`

- If <label> unspecified, defaults to `.udata_ovr`
- **Un-initialised** data in this segment is *overlayed* with all other data in `udata_ovr` segments of the *same name* <label>

Directives for Object File Memory Segments

UDATA_SHR: Begin Object File Un-initialised Data
Segment in Unbanked RAM

```
[<label>] udata_shr [<RAM_addr>]
```

- Use when generating relocatable object file
- If <label> unspecified, defaults to `.udata_shr`
- Declares a segment of Un-initialised data in unbanked RAM
- Starting address initialised to <RAM_addr>, or at link time if no address specified

Directives for (Relocatable) Memory Segments

IDATA: Begin an Object File Initialised Data Segment

`[<label>] idata [<RAM addr>]`

- Use when generating relocatable object file
- If <label> unspecified, defaults to `.idata`
- Location Counter initialised to <RAM_addr>, or at link time if no address specified
- Linker generates look-up table entry in ROM for each entry. User must add initialisation code to copy values from ROM to RAM. See `IDATA.ASM`

`initialisedGlobals:`

`idata`

`LimitL: dw 0`

`LimitH: dw D'300'`

`Gain: dw D'5'`

`Flags db 0`

`String db "Y-Axis",0`

Relocatable *Program Memory* Segments

Location of executable code:

- ***Absolute***: Use an `org` directive to locate code at an absolute address
- ***Relocatable***: Declare a code segment and allow the linker to calculate the address
- Valid address ranges specified in a *Linker Script File*

Relocatable *Data Memory* Segments

- **Data (variables) can be assigned to 1 of 5 segments:**
 - `udata`
 - `udata_acs` Each Un-initialised
 - `udata_ovr` Use the `RES` directive
 - `udata_shr`
 - `idata` Initialised (at least potentially...)
-
- The linker will place each of these in RAM, at locations specified by a linker file, `xxx.lkr`
 - If a linker file is not added to the project, the default generic linker file `18f452_g.lkr` is used

Un-initialised Data Memory Segments

- Data stored in any of these segments is not initialised
- Can only be accessed through:
 - Labels (variable names) declared in the segment
 - Indirect addressing
- `udata` – Un-initialised data, placed in RAM > 0x80
- `udata_acs` – access data, placed in Access RAM
- `udata_ovr` – overlaid data
 - Used for variables that can be placed at the same addresses because they exist at different, non-overlapping times
- `udata_shr` – shared data – placed in RAM that is not banked, or can be accessed in all banks
 - Not used in PIC18

`idata` – Initialised Data Memory Segment

- Data elements in `idata` are *initialised* – given initial values
- Use the `DB`, `DW`, or `DATA` directives
- **Question:** `idata` is a RAM segment, so where do the initial values come from?

`idata` – Initialised Data Memory Segment

- The linker generates and populates a table (the “`_cinit` table”) that contains an entry for each initialised data element
- Table begins with a 16-bit number `num_init` that stores the number of initialised data element
- Each table entry has *three* 32-bit integers that store
 - The `from` address in ROM (FLASH)
 - The `to` address in `idata` RAM
 - The `size` in bytes of the data element
- User code must copy each `from` to the corresponding `to` at run-time, but before the main code executes
- See the examples in `IDATA.asm` and `c18i.c`

Example – Data Segments

- From the .lst file generated by MPASM:

	00020	IDATA	
000000 00	00021	HistoryVector	DB 0
	00022		
	00023	UDATA	
000000	00024	InputGain	RES 1
000001	00025	OutputGain	RES 1
	00026		
	00027	Overlay1	UDATA_OVR
000000	00028	Temp1	RES 1
000001	00029	Temp2	RES 1
000002	00030	Temp3	RES 1
	00031		
	00032	Overlay2	UDATA_OVR
000000	00033	Hemp1	RES 1
000001	00034	Hemp2	RES 1

Name of this segment

Segment type

The Linker (MPLINK) and Librarian (MPLIB)

The Linker – MPLINK

- ***Locates code and data*** – Given relocatable object code and linker script, places code and data in memory
- ***Resolves addresses*** – calculates absolute addresses of external object modules
- ***Generates an executable*** – a .HEX file of specified format
- **Configures (software) *stack size* and location**
- **Identifies *address conflicts***
- **Produces *symbolic debug information*** – allows the use of symbols for variables, functions, rather than addresses.

MPLINK Inputs

- **.o** – Relocatable object files
- **.lib** – Collections of relocatable object files
 - Usually grouped in a modular fashion
 - Only used modules are linked into the executable
- **.lkr** – Linker script files tell the linker
 - What files to link
 - Range of valid memory addresses for a particular target

MPLINK Outputs

- **.hex – Binary executable file**
 - Intel HEX format / 8-bit split format / 32-bit HEX format
 - No debug information
- **.cof – Binary executable file in COFF (Common Object File Format)**
 - Also contains symbolic debug information
- **.map – Load map, showing memory use after linking**
 - Identify absolute addresses of globals, functions

Linker Script File (xxx.lkr)

```
// Sample linker command file for the PIC18F452 processor
// when used **with**** the MPLAB ICD2
```

Search
path

```
// Search for Libraries in the current directory.
LIBPATH .
```

Names of different
CODE segments

```
// CODEPAGE defined memory regions are in Program Memory, and are used for
// program code, constants (including constant strings), and the initial values
// of initialised variables.
```

CODE
#pragma code

CODEPAGE	NAME=vectors	START=0x000000	END=0x000029	PROTECTED
CODEPAGE	NAME=page	START=0x00002A	END=0x007DBF	
CODEPAGE	NAME=debug	START=0x007DC0	END=0x007FFF	PROTECTED
CODEPAGE	NAME=idlocs	START=0x200000	END=0x200007	PROTECTED
CODEPAGE	NAME=config	START=0x300000	END=0x30000D	PROTECTED
CODEPAGE	NAME=devid	START=0x3FFFFFFE	END=0x3FFFFFFF	PROTECTED
CODEPAGE	NAME=eedata	START=0xF00000	END=0xF000FF	PROTECTED

Only usable
by code that
requests it

Linker Script File (xxx.lkr) (continued)

Access RAM

```
// ACCESSBANK defined memory regions in Access RAM, used for data (variables).  
// DATABANK defined memory regions in Banked RAM, used for data (variables).  
// The names gpr0, gpr1, etc here are **arbitrary**.
```

```
ACCESSBANK NAME=accessram START=0x000 END=0x07F  
DATABANK NAME=gpr0 START=0x080 END=0x0FF  
DATABANK NAME=gpr1 START=0x100 END=0x1FF  
DATABANK NAME=gpr2 START=0x200 END=0x2FF  
DATABANK NAME=gpr3 START=0x300 END=0x3FF  
DATABANK NAME=gpr4 START=0x400 END=0x4FF  
DATABANK NAME=gpr5 START=0x500 END=0x5FF  
DATABANK NAME=dbgspr START=0x5F4 END=0x5FF PROTECTED  
ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFF PROTECTED
```

```
// Logical sections specify which of the memory regions defined above should  
// be used for a portion of relocatable code generated from a named section in  
// the source code. Each SECTION places a named section into a defined memory  
// region. Code sections are named using (for example) a  
// xxxx CODE directive in an assembly language source file OR  
// #pragma code directive in a C source file.
```

```
SECTION NAME=CONFIG ROM=config
```

Linker Usage

- **See MPLINK User's Manual for**
 - **Much more detail**
 - **Many examples**

The Librarian – MPLIB

- Allows construction & maintenance of object libraries
- Runs from the command line (DOS Window)

- Syntax is

`mplib [/q] /{ctdrx} Library [Member...]`

where

- `q`: Quiet mode
- `c`: Create Library with Member[s]
- `t`: List table showing Library members
- `d`: Delete Member[s] from Library
- `r`: Add/replace Member[s] in Library
- `x`: Extract Member[s] from Library