MPASM Assembler

with MPLINK and MPLIB

Linker and Librarian

What is <A> ?

- Assembly Language
  - Instructions for a µP 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'   ⇨   00001111 10101010
```

- Invariably, an assembler:
  - Has a set of *directives* that control assembler processing
  - Calculates relative addresses from instruction labels and variable names
- 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

MPASM Assembler Files

Input to Assembler:

- `.asm` **Assembly language source file**

Output from Assembler:

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

Assembler Listing File (`.lst`) Format

```
MPASM 5.57                        SAMPLE.ASM   8-25-2014  12:37:00


LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                      00001 ; Sample MPASM Source Code.
                      00002
                      00003         list p=18F452
  0000000B            00004 Dest    equu     0x0B
                      00005
                      00006         org     0x0000
000000                00007 Start:
000000 0E0A           00008         movlw   0x0A
000002 6E0B           00009         movwf   Dest
000004 EF?? F???      00010         goto    Start
                      00011
                      00012         org     0x0124
000124 EF?? F???      00013         goto    Start
                      00014
                      00015         end
```

Assembler Listing File (`.lst`) Format

```
MPASM 5.57                        SAMPLE.ASM   8-25-2014  12:37:00


SYMBOL TABLE
  LABEL                             VALUE

Dest                              0000000B
Start                             00000000
__18F452                          00000001

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

defined because we are building for the PIC18F452 processor

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

MPLINK Linker Files

Input to Linker:

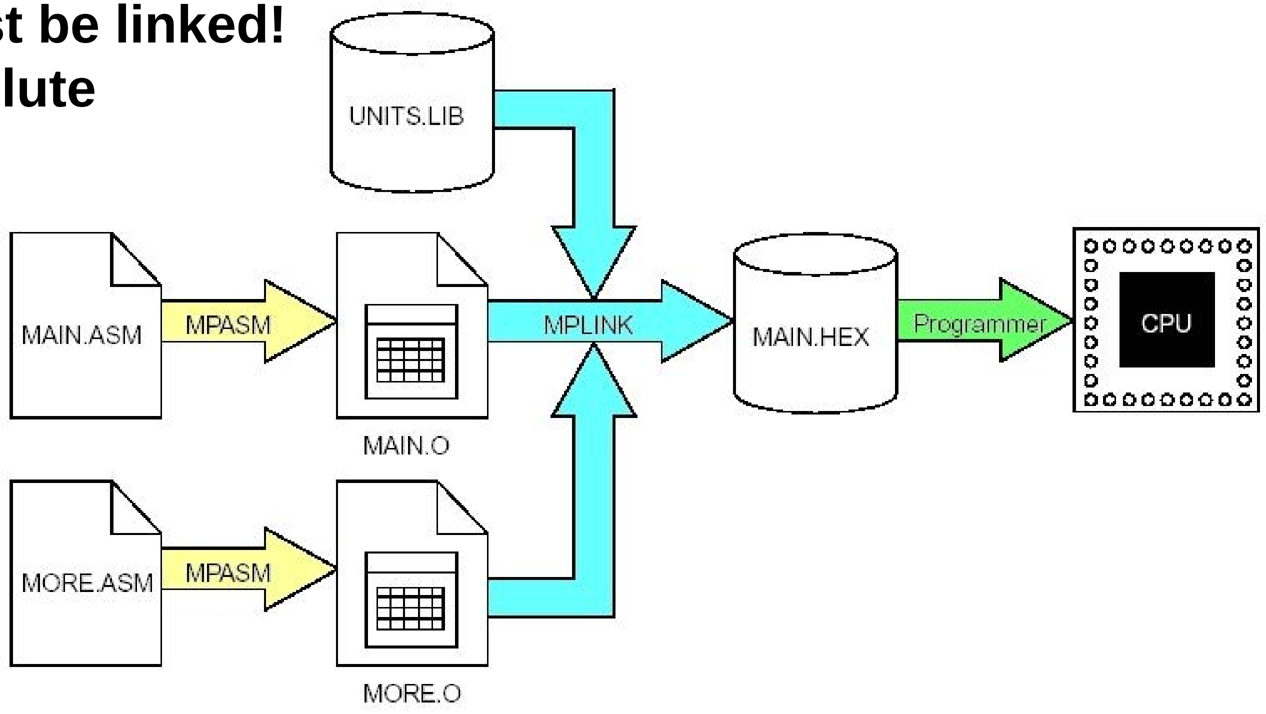
- `.o` Relocatable object file – or –
- `.lib` Relocatable object code in library file

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

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)



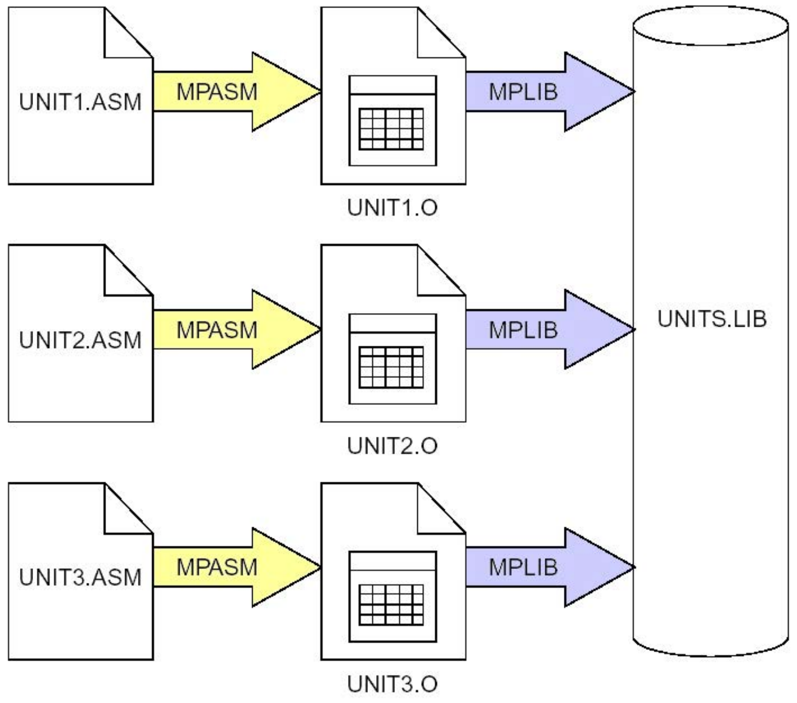
MPLAB X File Structure: Assembler and Linker Outputs

```
\---Example.X
    |   Source files: *.asm, *.inc
    |   Makefile
    |
    +---build      (Assembler Output – safe to delete)
    |   \---default
    |       +---debug
    |       |     Listing files *.lst
    |       |     Error files *.err
    |       |     Relocatable Object files *.o
    |       |
    |       \---production
    |             Listing files *.lst
    |             Error files *.err
    |             Relocatable Object files *.o
    |
    +---dist       (Linker Output – safe to delete)
    |   \---default
    |       +---debug
    |       |     Example.X.debug.cof
    |       |     Example.X.debug.hex
    |       |     Example.X.debug.map
    |       |
    |
    |       \---production
    |             Example.X.production.cof
    |             Example.X.production.hex
    |             Example.X.production.map
    |
```

```
    |
    |
    |
    |
    \---nbproject   (Project Configuration – don’t edit!)
        |   configurations.xml
        |   Makefile-default.mk
        |   Makefile-genesis.properties
        |   Makefile-impl.mk
        |   Makefile-local-default.mk
        |   Makefile-variables.mk
        |   Package-default.bash
        |   project.properties
        |   project.xml
        |
        \---private
                configurations.xml
                private.properties
```

Workflow – Librarian

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



The Assembler: MPASM

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
- An integral part of MPLAB X IDE

MPLAB X Help Files

- MPLAB X is Microchip’s (free) IDE
- MPLAB X v. 3.05 available from UoS website

- The on-line help files in MPLAB X are good
- Go to Help ⇨ Contents ⇨ MPASMX Toolsuite

Assembly Language 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**
  - **Start in column 1 (corollary: anything starting in column 1 is a label)**
  - **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 (e.g. `movlw`)**
  - **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`	(default)
▪ **Decimal**	`D’163’`	Note – all the same number,
▪ **Octal:**	`O’243’`	but different radixes!
▪ **Binary:**	`B’10100011’`	

- **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 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
- They are not case sensitive
- There are many arithmetic and logic operators that can be used to construct directive expressions
  - For example: +, -, *, /, <<, ==, <=, ~, !, &, ||, etc…
  - See precedence table in MPASM User’s Guide

Assembler Directives

- There are six types of assembler directives, for
  1. Assembler Control
  2. Conditional Assembly
  3. Data Definition
  4. Listing Control
  5. Object File Control
  6. Macro Definition

1. Assembler Control Directives

- Defining which processor we are building for
- Defining assembler error reporting level
- Defining symbolic names
- Including files

Assembler Control – Configuration

PROCESSOR: Defines the build target processor

```
processor <processor>
```

```
processor 18f452
```

RADIX: Specify default radix (base)

```
radix <default_radix>
```

```
radix hex
```

- Options are **`hex`**, **`dec`**, **`oct`**
- Radix defaults to **`hex`** if not specified

ERRORLEVEL: Set diagnostic message level

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

```
errorlevel 0
```

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

Assembler Control – Symbols

EQU: Defines a symbolic label for a constant

```
<label> equ <expr>
```

- `expr` is a number
- See examples in `p18f452.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’
constant MASK = ~(0xAA)
flags &= MASK
```

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*
- For PIC18 `<expr>` must be an even number
- The location counter value at assembly time is equivalent to the PC value at run time

- Cannot be used when generating a relocatable object file
- Use CODE, UDATA, UDATA_ACS, IDATA directives instead (see later)

```
    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\filename”
#include  path\filename
```

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

```
; get register symbols, etc
#include p18f452.inc

; define configuration bits
#include configReg.inc
```

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”

; Load TBLPTR with the address of ‘I’
   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, F, BANKED       ; 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

2. Conditional Assembly Directives

- Permit sections of code to be conditionally assembled
- These are active only during assembly – they are not active at run time
- Similar to C language preprocessor directives – e.g.

```
#ifdef TESTING
// some C code here
#endif
```

Conditional Assembly – `if-else-endif`

We 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

```
if rate < 50
    INCF  speed, F
else
    DECF  speed, F
endif
```

Conditional Assembly – While

WHILE: Loop while `<expr>` is TRUE

ENDW: End of a WHILE loop

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

- `<expr>` is evaluated and a non-zero value is interpreted as logically TRUE
- `<assembly_code>` cannot exceed 100 lines
- Cannot 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 ⇨ **`end`**
ENDIF:	**End of a conditional block**	**`if`** ⬄ **`endif`**
ENDW:	**End of a while loop**	**`while`** ⬄ **`endw`**
ENDM:	**End of a macro definition**	**`macro`** ⬄ **`endm`**
ENDC:	**End an automatic constant block** (**`cblock`** defines a list of named constants)	**`cblock`** ⬄ **`endc`**

3. Data Definition Directives

- **Control memory allocation and symbol definition**
- **This is how to define named “variables” in RAM and named constant values in FLASH**

Data Definition – Integers

DB: Declare data of one byte (in ROM or RAM)
DW: Declare data of one word (in ROM or RAM)
DE: Declare EEPROM data of one byte

```
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** and **code_pack** directives
  - **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 – 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`**

```
data    ‘C’        ; one character
data    “Sharp”    ; string
Numbers:
data    1, 2, 7    ; some numbers
```

FILL: Fill memory block with value

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

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

```
fill    0x5555, D’10’
fill    (nop), 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 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 – µC Configuration

PROCESSOR: Set processor type

`processor <processsor_type>`

```
PROCESSOR 18f452
```

CONFIG: Set processor configuration bits

`config <bit>=<value>`

`__config` is deprecated

- Sets the configuration bits
- Processor must previously have been declared
- Best practice: use to over-ride MPLAB X config bit settings. See `configReg.inc` on server
- See usage and definitions in `p18f452.inc`

```
#include configReg.inc   ; or

; Configuration Register 1H
; Oscillator switch disabled,
; EC oscillator.
CONFIG   OSCS=OFF, OSC=EC
;
; etc, ...
```

__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'
```

4. Listing Control Directives

- Directives to control the content and format of the assembler listing file xxxxx.lst

4. Listing Control Directives

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
```

Listing Control Directives

LIST: **Turn on listing, with options**

NOLIST: **Turn off listing**

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

- `list`, with no options, turns listing on
- Options (14 of them) control various listing settings – see Assembler Manual / help
- `nolist` turns off listing

```
; suppress listing of symbols, etc
nolist
include p18f452.inc

; turn listing back on
list
```

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

Listing Control Directives

ERROR: Issue a user-defined error message

MESSG: Create user-defined message

```
messg “message_text”
error “error_text”
```

- Both print user-defined messages

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

```
#include p18f452.inc

variable baudrate
baudrate set D'5600' ; required baud rate

if (baudrate!=D'1200')&&(baudrate!=D'2400')&&
   (baudrate!=D'4800')&&(baudrate!=D'9600')&&
   (baudrate!=D'19200')
  error "Selected baud rate is not supported"
  messg "only baud rates 1200,2400,4800, "&&
        "9600 & 19200 Hz are supported"
endif
```

Relocatable Object Files

Relocatable Object Files

- **Assembled (or compiled) object files xxxxx.o 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**

5. Directives for Relocatable Object Code

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 using/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

```
; Subroutine is called from one
; assembly language file
        extern Subroutine
        call Subroutine
```

GLOBAL: Export a label to the linker

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

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

```
; Subroutine is defined in
; a different file
       global Subroutine
Subroutine: code
       ; body of subroutine
       return
```

Relocatable Program Memory Segments

Location of executable code (or constant values) in ROM

- ***Absolute*: Use an `org` directive to locate code at an absolute address**
- ***Relocatable*: Declare a `code` segment (or a `code_pack` segment) and allow the linker to calculate the address**

- **Valid address ranges specified in a *Linker Script File***

Directives for Object Code Memory Segments

CODE: Begin an executable code segment (or constants stored in program memory)

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

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

```
; Executable code
RST     code 0x00
        goto start
```

```
; Padded data – append padding
; byte of 0 to odd number or bytes
padded: code
       DB 1, 2, 3
       DB 4, 5
```

CODE_PACK: Begin packed code segment (constants stored efficiently in program memory)

```
[<label>] code_pack[<ROM_addr>]
```

- Used to place constant data (one byte per byte) into FLASH memory (use **db**)
- Use with **de** to place constant data into EEPROM

```
; Packed data – no padding bytes
; appended
packed: code_pack 0x1F0
       DB 1, 2, 3
       DB 4, 5
```

Relocatable Data Memory Segments

- Data (variables) can be assigned to 1 of 5 segments:
  - `udata`
  - `udata_acs` — Each Un-initialised - use the
  - `udata_ovr` — `RES` directive
  - `idata`
    `idata _acs` — Initialised (at least potentially…) – use `DB`, `DW`, etc. directives
- 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**

Directives for Object Code Memory Segments

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 (banked) RAM
        udata
aVariable:  res  1
```

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
```

Directives for Object Code Memory Segments

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>`

`idata` – Initialised Data Memory Segment

- Data elements in `idata` or `idata_acs` can be *initialised* – that is, given initial values
- Use the `DB`, `DW`, or `DATA` directives to
  - a) reserve memory and
  - b) specify initial values

- Question: `idata` is a RAM segment (volatile), so where do the initial values come from?

Directives for Object Code 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 using the "**`_cinit table`**". See the file **`IDATA.ASM`** for a good example

```
initialisedGlobals:
        idata
LimitL: dw  0
LimitH: dw  D'300'
Gain:   dw  D'5'
Flags   db  0
String  db  "Y-Axis",0
```

`idata` – Initialised Data Memory Segment

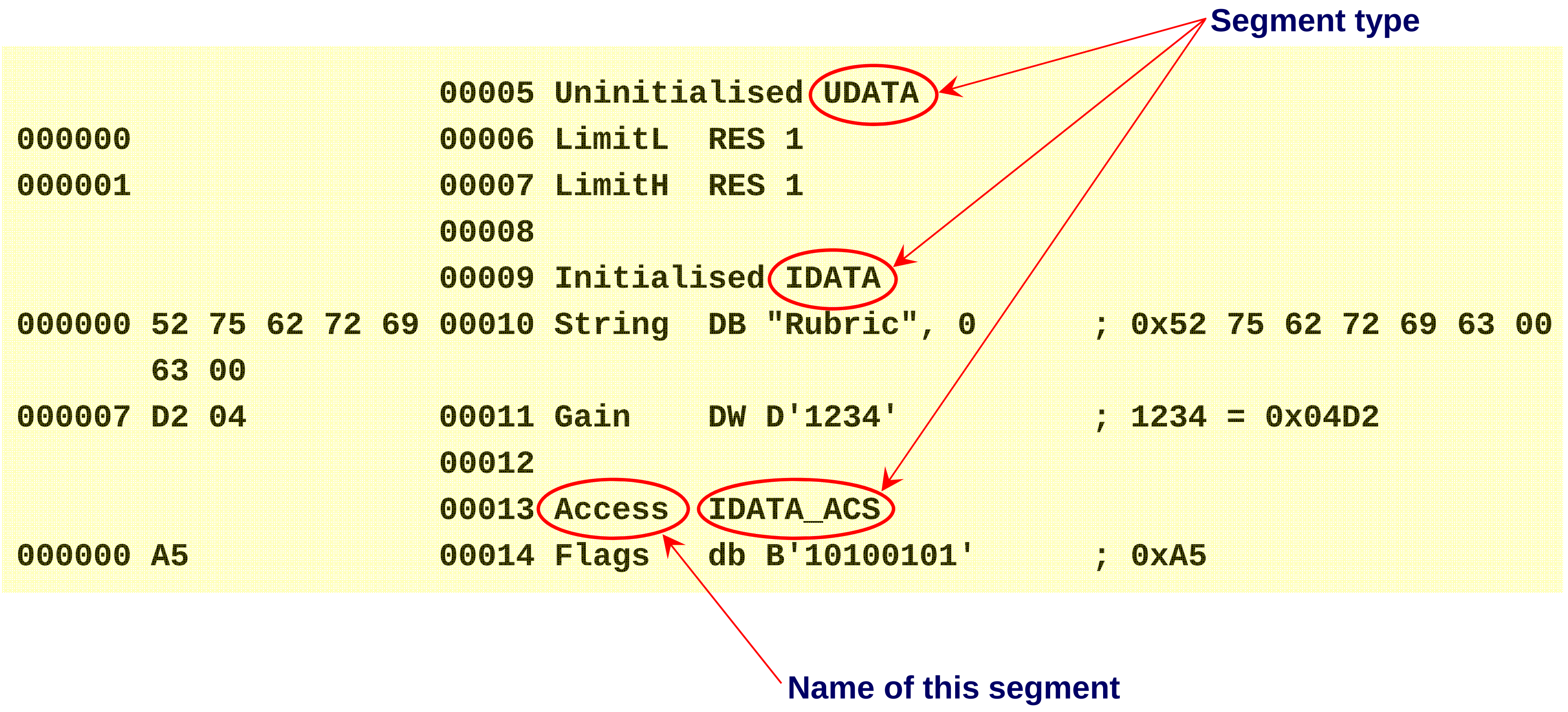
- The linker generates and populates a table (the "`_cinit` table") in ROM that contains an entry for each initialised data element
- Table begins with a 16-bit number `num_init` that stores the number of initialised data element
- Each table entry has *three* 32-bit integers that store
  - The `from` address in ROM (FLASH)
  - The `to` address in `idata` or `idata_acs` RAM
  - The `size` in bytes of the data element

- User code must copy each `from` byte to the corresponding `to` at run-time, but before the main code (assembler or C) executes

- See the examples in `IDATA.asm` and `c18i.c`

Example – Data Segments

- From the `.lst` file generated by MPASM



```
                        00005 Uninitialised UDATA
000000                  00006 LimitL   RES 1
000001                  00007 LimitH   RES 1
                        00008
                        00009 Initialised IDATA
000000 52 75 62 72 69   00010 String   DB "Rubric", 0      ; 0x52 75 62 72 69 63 00
       63 00
000007 D2 04            00011 Gain     DW D'1234'          ; 1234 = 0x04D2
                        00012
                        00013 Access  IDATA_ACS
000000 A5               00014 Flags    db B'10100101'      ; 0xA5
```

Segment type
Name of this segment

Example – Data Segments

- From the `.map` file generated by MPLINK

Section `.cinit` contains the `_cinit` table

`.cinit` section starts at `0x000008`

```
Section Info
      Section       Type    Address   Location Size(Bytes)
    ---------  ---------  ---------  ---------  ---------
       .org_0       code   0x000000    program   0x000008
       .cinit    romdata   0x000008    program   0x00001a
Initialised_i    romdata   0x000022    program   0x000009
     Access_i    romdata   0x00002b    program   0x000001
       .org_1       code   0x000124    program   0x000004
       Access      idata   0x000000       data   0x000001
  Initialised      idata   0x000080       data   0x000009
Uninitialised      udata   0x000089       data   0x000002
```

Section `Initialised_i` contains the initial values of initialised variables

Section `Initialised` contains the initialised variables

Example – Data Segments

- Contents of Program Memory

```
movlw   0x0A
movwf   DEST
goto    START
```

start of _cinit table

```
Addr   00     02     04     06     08     0A     0C     0E
0000   0E0A   6E0B   EF00   F000   0002   002B   0000   0000     ...n.... ..+.....
0010   0000   0001   0000   0022   0000   0080   0000   0009     ......". ........
0020   0000   7552   7262   6369   D200   A504   FFFF   FFFF     ..Rubric ........
              u R    r b    c i    /0
```

initial value of `Gain` = `0x04D2`

initial value of `Flags` = `0xA5`

- _cinit Table

```
0002
0000 002B    0000 0000    0000 0001
0000 0022    0000 0080    0000 0009
```

num_init = 2

copy 1 byte from `0x00002B` (ROM)to `0x000` (RAM)

copy 9 bytes from `0x000022` (ROM)to `0x080` (RAM)

MPASM Macro Language

A simple form of preprocessor that allows for 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 symbolic 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`, `expand` / `noexpand`

6. Directives for Macro Definition

- **Control execution and data allocation within macros**
  - **`macro` - Declare macro Definition**
  - **`exitm` - Exit from a macro (stop expansion; exit to the end`m`)**
  - **end`m` - End a macro definition**
  - **expand - Expand macro listing**
  - **noexpand - Turn off macro expansion**
  - **`local` - Declare a local macro variable**

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
size    equ 20

m_buffer:
        macro   size
        local   len, label
len     set     size
label   res     len
len     set     len - size
endm
```

Example – Macro Definition

- **Macro definition is**

```
#include “p18f452.inc”
;
; Compare register aReg to a constant aConst and
; jump to aDest if register value >= constant.
;
mCmpJge:     macro aReg, aConstant, aDestination
             movlw    aConstant
             subwf    aReg, W
             btfsc    status, carry
             goto     aDestination
             endm
```

Example – Macro Invocation

- **When invoked ("called") by:**

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

the macro `mCmpJge` will produce (expand to):

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

The Linker (MPLINK)

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 (in C)
- 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`)

Search path

Names of different CODE segments

Only usable by code that requests it

CODE
`#pragma code`

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

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

// 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.

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=0x3FFFFE      END=0x3FFFFF       PROTECTED
CODEPAGE   NAME=eedata    START=0xF00000      END=0xF000FF       PROTECTED
```

Linker Script File (`xxx.lkr`) (continued)

```
// 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, grp1, 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=0x5F3
DATABANK   NAME=dbgspr     START=0x5F4          END=0x5FF          PROTECTED
ACCESSBANK NAME=accesssfr  START=0xF80          END=0xFFF          PROTECTED
```

Access RAM

Linker Script File (`xxx.lkr`) (continued)

```
// 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 directive defines a name for previously define
// memory region. This defined name can be referenced from the user’s code.
SECTION    NAME=MAINCODE     ROM=page
SECTION    NAME=PAGE2        RAM=gpr2


// Code sections are referred to in user code using (for example)
// in assembler:
// MAINCODE CODE
// or in C:
// #pragma idata PAGE2
```

these names used in code

Linker Usage

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

The Librarian (MPLIB)

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`