11030 HTC

Using HI-TECH PRO C Compiler Introducing HI-TIDE™ & C-Wiz

Class Objectives

To become familiar with:

- **HI-TIDE™ and C-Wiz**
- **PRO Version of the Compiler**
  - Compilation workflow
  - Source code differences
- **New STD Compiler Features**

Class Agenda

- **Compiler Overview**
- **Demonstration of HI-TIDE & C-Wiz**
- **Data Types & Qualifiers**
- **Diagnostic Files**
  - Lab 1: Using the diagnostic files
- **Interrupts**
  - Lab 2: Using interrupts

Class Agenda (cont.)

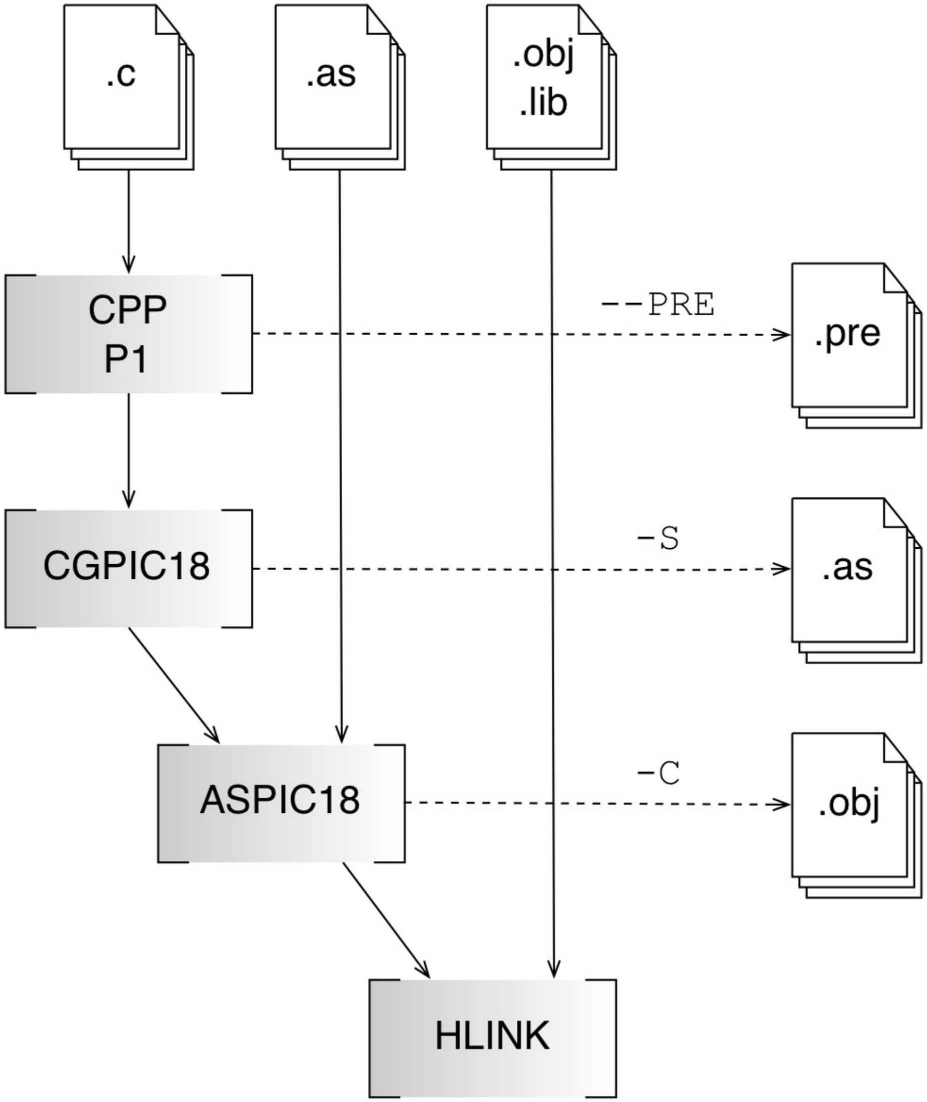
- **Library & Compiler-Generated Code**
  - Lab 3: Defining power-up code
- **Psects and the Linker**
- **HI-TECH Assembly**
  - Lab 4: Placing code at a specific location

- **Compiler, HI-TIDE & C-Wiz**
- Data Types & Qualifiers
- Diagnostic Files & Options
- Interrupts
- Library & Compiler-Generated Code
- Psects & the Linker
- HI-TECH Assembly

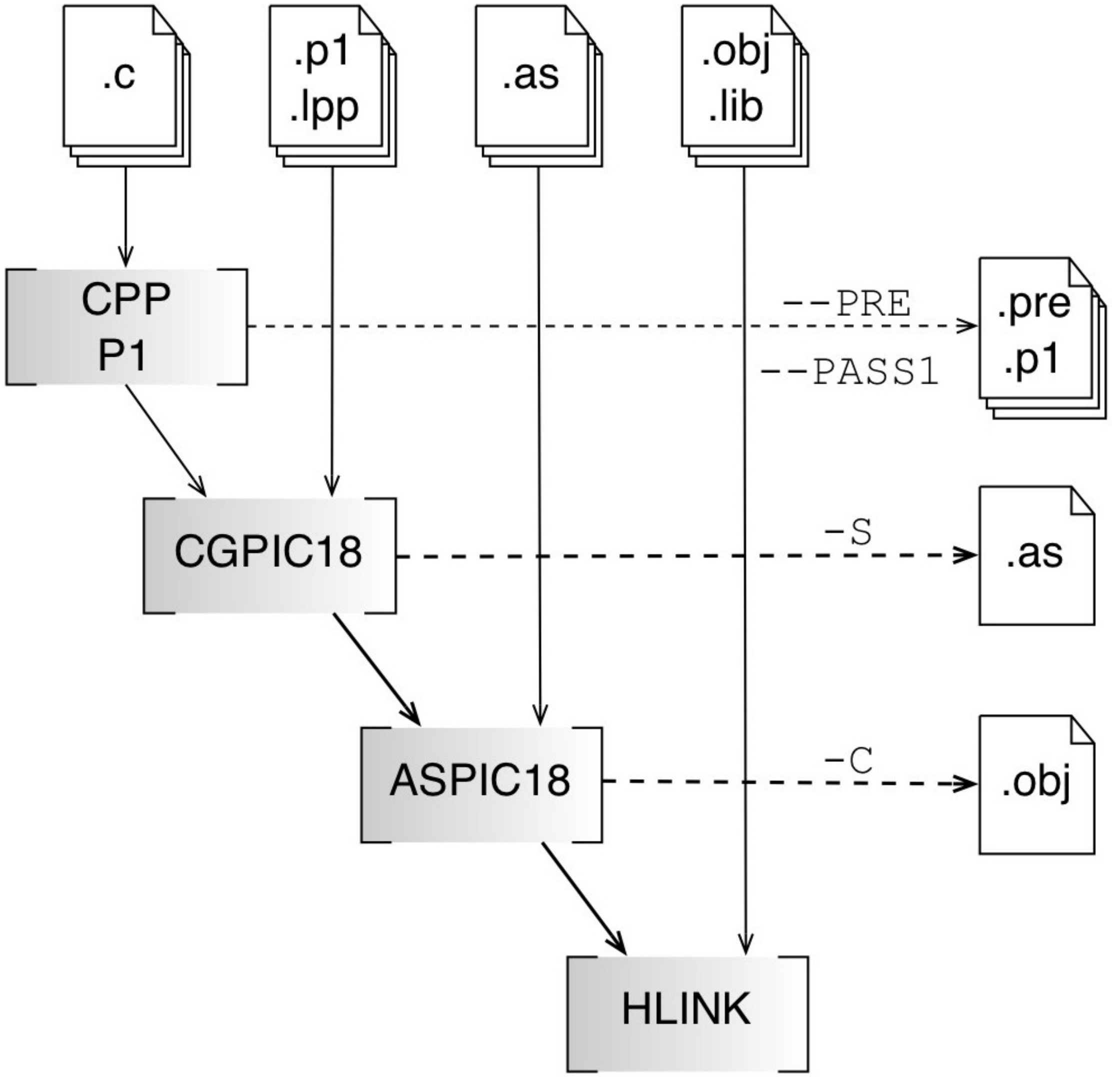
Compiler Overview

- **HI-TECH PIC18 Compiler consists of Several Applications:**
  - CPP & P1 C preprocessor & parser
  - CGPIC18 code generator
  - ASPIC18 assembler
  - HLINK linker
  - OBJTOHEX, CROMWELL & HEXMATE output utilities

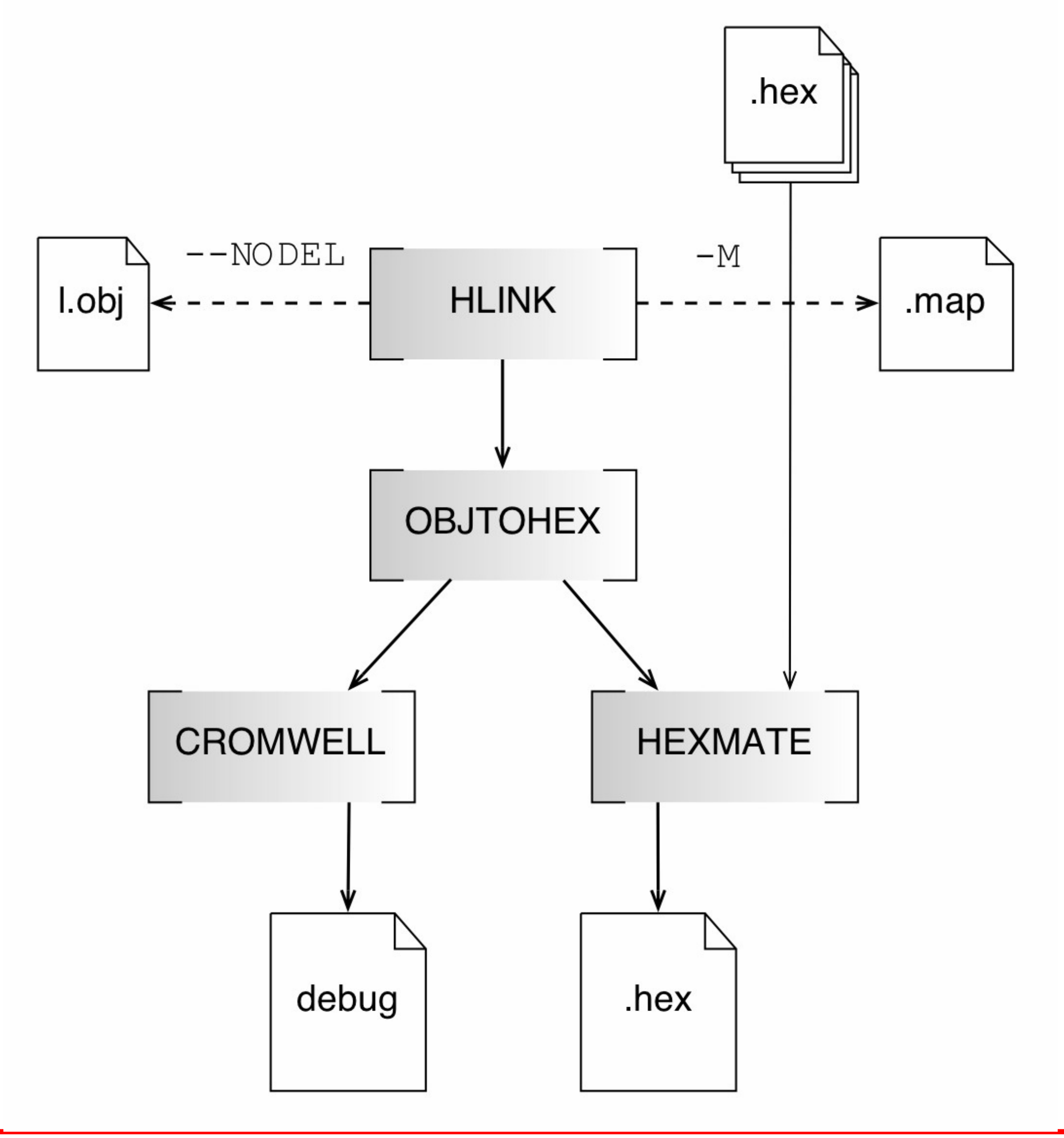
STD compiler input sequence



PRO compiler input sequence



Compiler output sequence



Command Line Driver

- **Driver:** `PICC18`
- **Calls Appropriate Applications based on Input File Extension:**

`.c`	`.as`	**C and assembly source files**
`.obj`	`.p1`	**Relocatable & p-code object files**
`.lib`	`.lpp`	**Object & p-code library files**
`.hex`		**Intel® HEX files**

Compiler Messaging

- Applications Report via a Driver-Controlled System

```
ts002.c: 159: (762) constant truncated
when assigned to bitfield (warning)
```

- Driver Options Available to:
  - Adjust format
  - Select language
  - Disable warnings

Demonstration

- **Getting Started with HI-TIDE**
  - Getting help
  - Creating projects
  - Debugging
- **Getting Started with C-Wiz**

- Compiler, HI-TIDE & C-Wiz
- **Data Types & Qualifiers**
- Diagnostic Files & Options
- Interrupts
- Library & Compiler-Generated Code
- Psects & the Linker
- HI-TECH Assembly

Data Types Supported

- **Standard Arithmetic Types**
  - `char` is `unsigned` by default
    - **Use** `--char=signed` **to change**
  - `double` types 24 bits by default
    - **Use** `--double=32` **to specify 32 bits**
- **24-bit `short` `long` integral type**
- **`bit` type used for boolean values**
  - 8 `bit` variables packed per byte

Data Types (cont.)

- Bit addresses used in diagnostic files
- `bit` variables cannot be `auto`

How can you define a `bit` variable with scope only in a function?

```
static bit flag;
```

- Integral conversion to `bit` type is via truncation

Standard Qualifiers Supported

- **`const` objects**
  - Read-only
  - Stored in program space
- **`volatile` objects**
  - Value may change between reads due to external modification
  - Optimizer won’t remove redundant accesses

Standard Qualifiers (cont.)

- Compiler will attempt atomic access
  - **Modify value in one instruction**
- Should be used for:
  - **Variables mapped over registers modified by hardware**
  - **Registers whose value translates to an electrical signal**
  - **Variables modified by interrupt routines**

HI-TECH Specific Qualifiers

- `near` **Objects**
  - Place in access bank
- `far` **Objects**
  - Place in program memory space
- `persistent` **Objects**
  - Not cleared by run-time start-up code

Absolute Objects

- Primarily Intended to Map Variables over an SFR, e.g.

```
volatile near unsigned char TOSH @ 0xFFE;
```

- Memory is Automatically Allocated by the Code Generator (CGEN)
- Header Files contain Absolute Variable Definitions for SFRs:

```
#include <htc.h>
```

Absolute Objects (cont.)

– Const objects and functions can be made absolute using similar construct

Pointers

- **Pointers to Data and Functions Supported**
  - HI-TECH specific qualifiers required to indicate pointer extent
  - Size and extent are determined from pointer usage
  - Standard qualifiers should still be used for `const` or `volatile` objects

Pointers (cont.)

```
near int ni;
int      i, j;
far int  fi;
int *    ip;

void main(void) {
   ip = &ni; // ip 1/2 bytes wide
   j  = *ip;
   ip = &i;  // ip 2/2 bytes wide
   j += *ip;
   ip = &fi; // ip 3/2 bytes wide
   j += *ip; // read from wrong loc
}
```

University of Microchip
U of M
MASTERs 2007

Question

Is it legal to qualify a variable both `const` and `volatile`? If not, why not; if so, what does it mean?

Yes. It means that it can only be read by the program, but that its value may change by other means.

Question

How would you define a read-only pointer variable that points to a `volatile` character variable?

```
volatile char * const my_pointer;
```

Memory Allocation

- **The CGEN can either:**
  - allocate an address to an object; or
  - place output in a named block (psect program section) for linker to position
- **Variables Allocated by CGEN are then Treated as Absolutes**

Variable Allocation

- **All variables are allocated memory by the CGEN, except:**
  - `const` objects
  - Initialized variables (`data` psect)
  - `auto`/parameter objects
- **CGEN allocates memory for absolute variables; remainder allocated by the linker**

Variable Allocation (cont.)

- `auto` & parameter variables form a block (APB) for each function
- The linker overlays APBs of functions not concurrently active
- The entire program’s APB is contained within one psect (`param`, `param0`, `param1`…)

- Compiler, HI-TIDE & C-Wiz
- Data Types & Qualifiers
- **Diagnostic Files & Options**
- Interrupts & Runtime Startup
- Library & Compiler-Generated Code
- Psects & the Linker
- HI-TECH Assembly

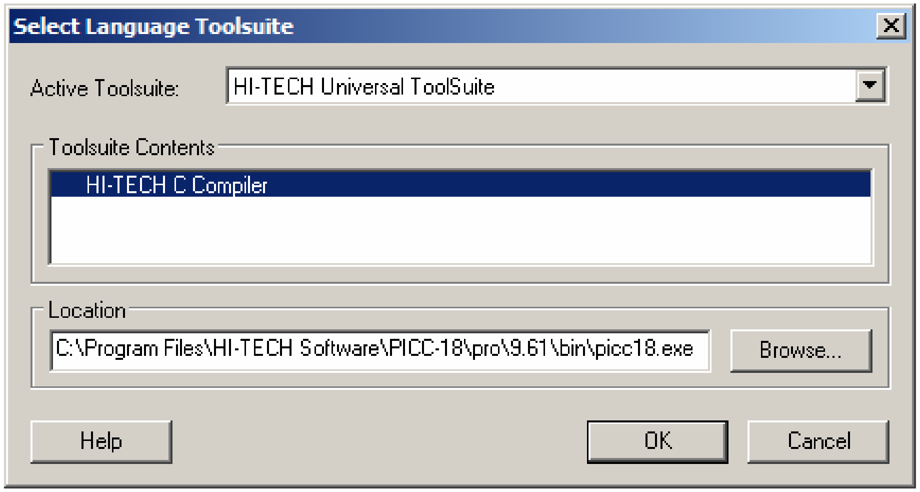
Useful General Options

`--help`	Show help
`--chipinfo`	List supported chips
`-V`	Show full command lines
`--msgdisable`	Control compiler messages
`--lang`	Select message language
`-D`	Define preprocessor macro
`--emi`	External Interface mode

Useful Debug Options

`--debugger`	Select debugger
`--opt`	Control optimizers
`--asmlist`	Generate assembly list file
`-M`	Generate map file
`--pre`	Stop after preprocessing
`--pass1`	Stop after parsing
`-S`	Stop after code generation
`-C`	Stop after assembling

MPLAB® IDE Project Setup



Select Language Toolsuite

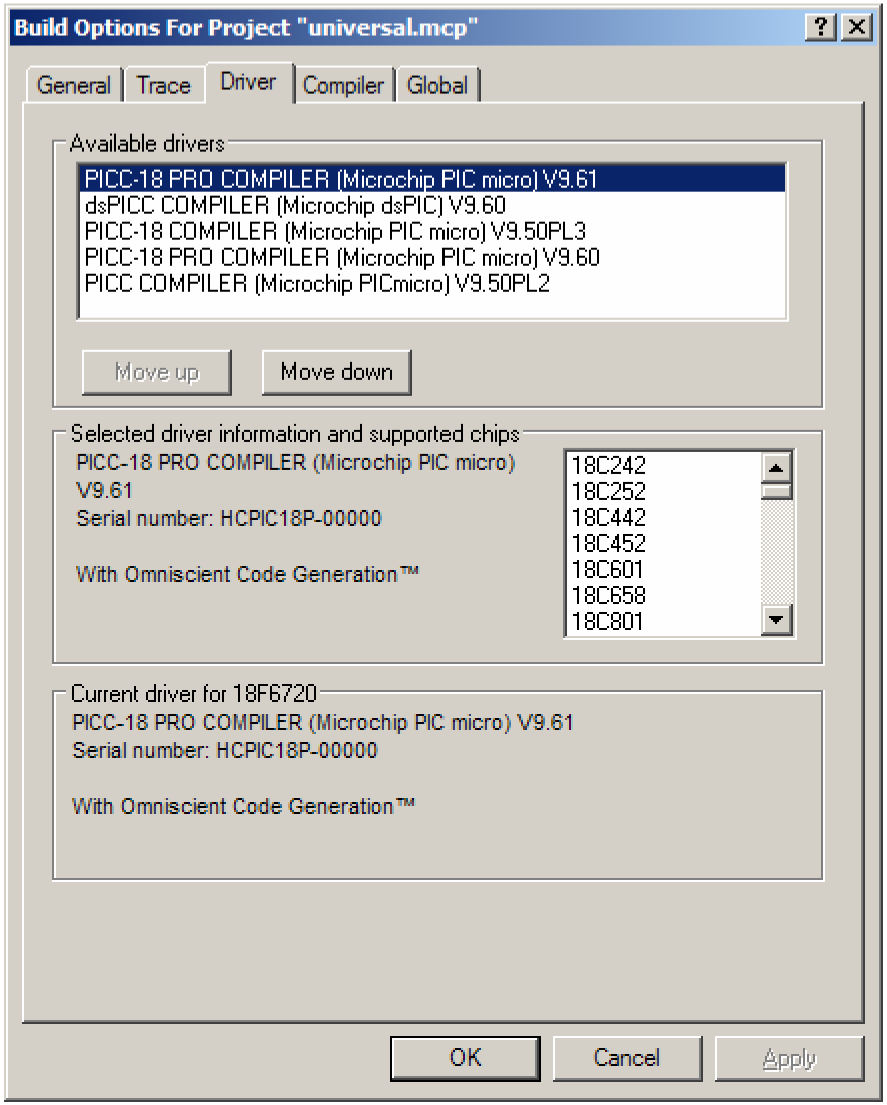
Active Toolsuite: HI-TECH Universal ToolSuite

Toolsuite Contents

HI-TECH C Compiler

Location

C:\Program Files\HI-TECH Software\PICC-18\pro\9.61\bin\picc18.exe



Build Options For Project "universal.mcp"

Available drivers

PICC-18 PRO COMPILER (Microchip PIC micro) V9.61
dsPICC COMPILER (Microchip dsPIC) V9.60
PICC-18 COMPILER (Microchip PIC micro) V9.50PL3
PICC-18 PRO COMPILER (Microchip PIC micro) V9.60
PICC COMPILER (Microchip PICmicro) V9.50PL2

Selected driver information and supported chips

PICC-18 PRO COMPILER (Microchip PIC micro)
V9.61
Serial number: HCPIC18P-00000

With Omniscient Code Generation™

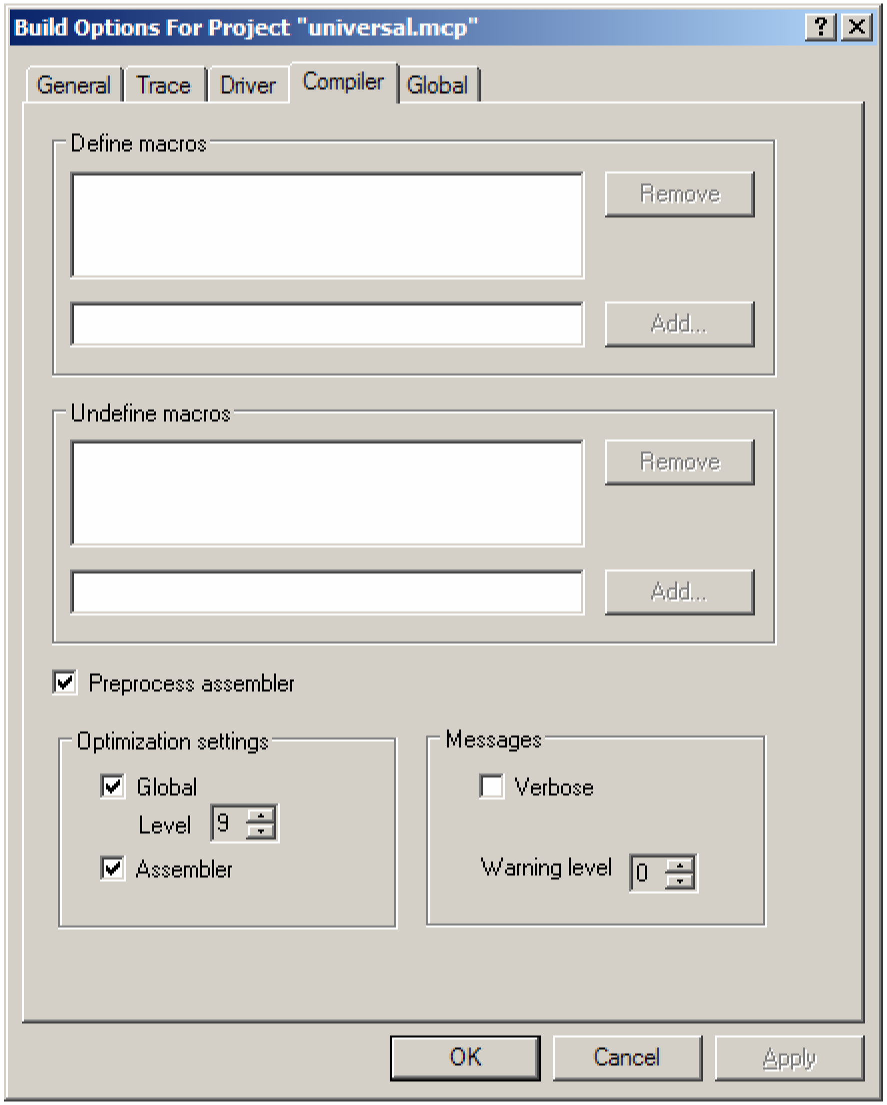
18C242
18C252
18C442
18C452
18C601
18C658
18C801

Current driver for 18F6720

PICC-18 PRO COMPILER (Microchip PIC micro) V9.61
Serial number: HCPIC18P-00000

With Omniscient Code Generation™

MPLAB® IDE Options Dialog



Build Options For Project "universal.mcp"

Define macros

Undefine macros

☑ Preprocess assembler

Optimization settings

☑ Global

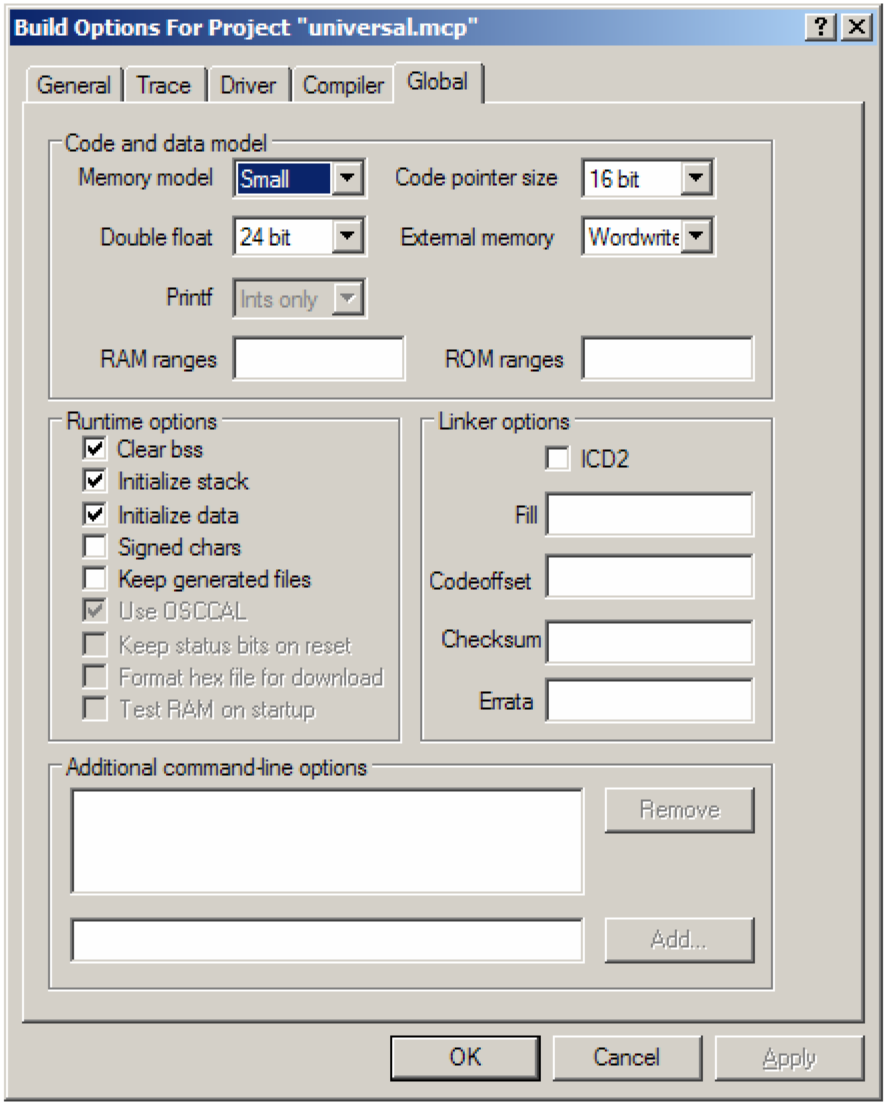
Level 9

☑ Assembler

Messages

☐ Verbose

Warning level 0



Build Options For Project "universal.mcp"

Code and data model

Memory model: Small

Code pointer size: 16 bit

Double float: 24 bit

External memory: Wordwrite

Printf: Ints only

RAM ranges

ROM ranges

Runtime options

☑ Clear bss

☑ Initialize stack

☑ Initialize data

☐ Signed chars

☐ Keep generated files

☑ Use OSCCAL

☐ Keep status bits on reset

☐ Format hex file for download

☐ Test RAM on startup

Linker options

☐ ICD2

Fill

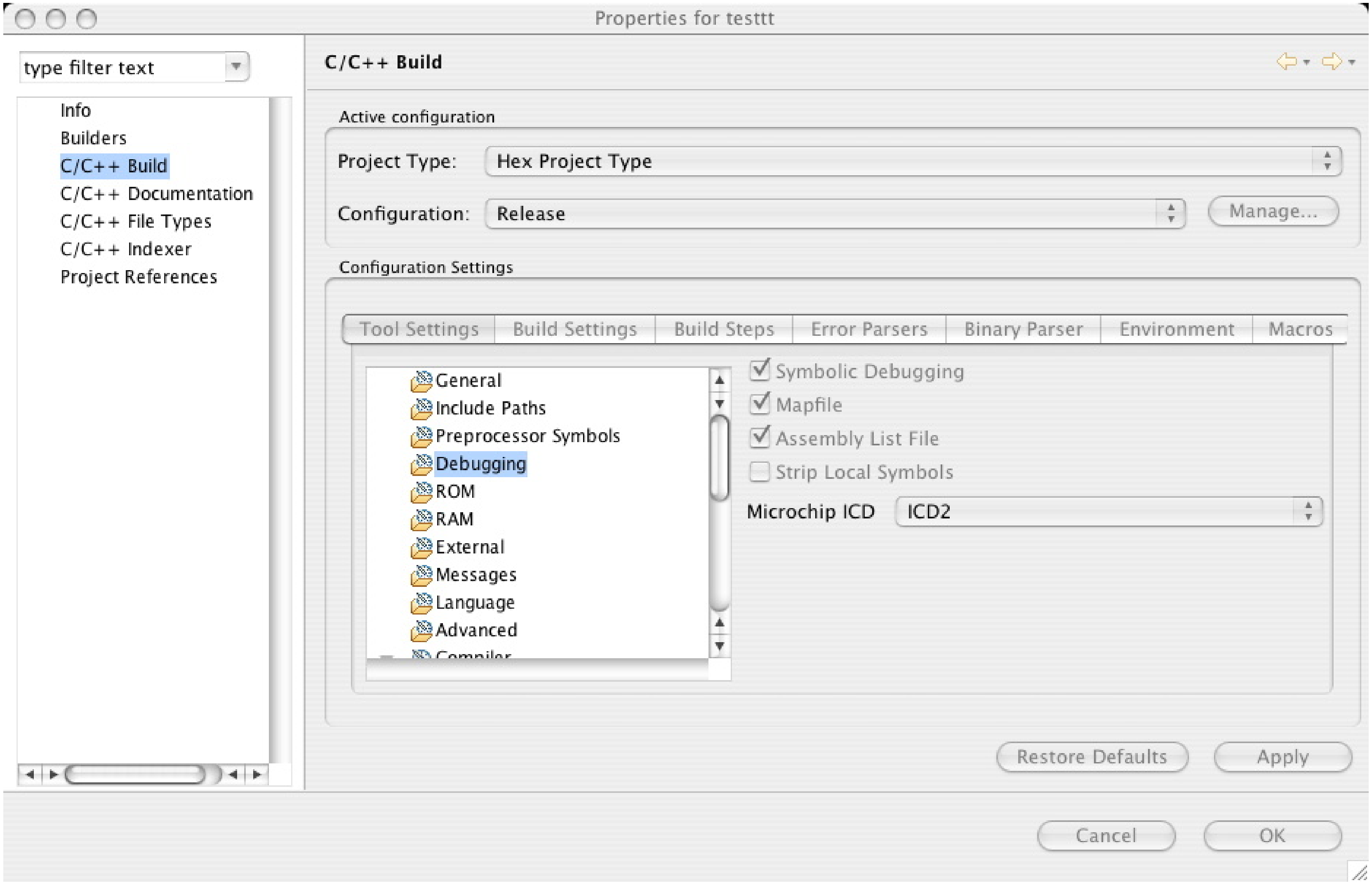
Codeoffset

Checksum

Errata

Additional command-line options

HI-TIDE Options Dialog



Properties for testttt

C/C++ Build

Active configuration

Project Type: Hex Project Type

Configuration: Release

Configuration Settings

☑ Symbolic Debugging

☑ Mapfile

☑ Assembly List File

□ Strip Local Symbols

Microchip ICD: ICD2

HEXMATE Control

`--fill`	**Fill unused memory**
`--checksum`	**Calculate and insert a checksum value**
`--serial`	**Insert bytes (serial number)**

- **Other HEXMATE Features:**
  - Merge Intel HEX files
  - Search for HEX codes and optionally replace with new codes

Controlling Messages

- `#pragma warning` allows control over individual errors & warnings

`disable` *`list`*	disable messages
`enable` *`list`*	enable messages
`push`	save current state
`pop`	retrieve previous state
`warning` *`list`*	make messages warning
`error` *`list`*	make messages error

Question

How could you easily send your source code to a colleague when the C source files include header files located in many folders?

Preprocess the source files using the `--pre` option. This includes the header files so they are contained within the output file.

Understanding List Files

- **The Assembler List Files show:**
  - The C or assembly source
  - The generated assembly code
  - Assembler directives
  - Absolute addresses determined by the linker
  - The module’s symbol table

List Files (cont.)

```
 96                                PSECT    text
 97  003FEA               _main:
 98  003FEA  FFFF                  DW       0FFFFh
 99  003FEC  D008                  goto     f22
100  003FEE               f21:
101                                GLOBAL   _foo
102                   ;main.c: 3:  void main(void)
103  003FEE  0E01                  movlw    01h
104  003FF0  CFE8 F5FF             movff    wreg,_foo
105                   ;main.c: 7:  foo++;
106  003FF4  0105                  movlb    _foo >> 8
107  003FF8  2BFF                  incf     _foo&0ffh,b
108                   ;main.c: 8:  }
```

Symbol Translations

C code symbols	Assembler mapping
`int gi;`	`_gi`
`void func`	`_func`
`(int pi)`	`?_func+0, ?_func+1`
`{`	
`int i;`	`??_func+0, ??_func+1`
`}`	`?a_func+0, ?a_func+1`

- Most C Definitions Map to an Assembler Label: `symbol:`

List Files (cont.)

- **Be Aware of the Following:**
  - A C list file with "`.lst`" extension is produced if `--asmlist` is not used
  - The assembler optimizer omits some assembler directives in the listing file
  - Absolute addresses are only shown if the linker runs to completion
    - **' marks show unresolved values**

List Files (cont.)

```
 96                             PSECT   text
 97  000000'             _main:
 98  000000'  FFFF              DW      0FFFFh
 99  000002'  D008              goto    f22
100  000004'             f21:
101                             GLOBAL  _foo
102               ;main.c: 3: void main(void)
103  000004'  0E01              movlw   01h
104  000006'  CFE8 F000'        movff   wreg,_foo
105               ;main.c: 7: foo++;
106  00000A'  0100'             movlb   _foo >> 8
107  00000E'  2B00'             incf    _foo&0ffh,b
108               ;main.c: 8: }
```

Understanding the Map File

- **The Map File consists of:**
  - The options used by the linker
  - The call graph
  - Psects defined by each module
  - Psect summary listed by class
  - Unused class memory locations
  - The program symbol table

The Map File (cont.)

```
HI-TECH Software PICC-18 Compiler V9.50

Linker command line:

--edf=C:\PROgram Files\HI-TECH
Software\PICC-18\9.50\dat\en_msgs.txt \
-h+main.sym -z -Q18F452 -ol.obj -Mmain.map
-ver=PICC-18#V9.50 \
-ACODE=00h-03FFFhx2 -ARAM=00h-0FFhx6 -
ABIGRAM=00h-05FFh -ACOMRAM=00h-07Fh \
-ANVRAM=0500h-05FFh –preset_vec=0h\
```

Call Graph Details

- **Indentation shows Call Hierarchy and Approximate Stack Usage**
  - Actual stack usage may be higher, due to interrupts, or lower due to optimizations
  - Indirect calls & parameters involving function calls show extra levels
- **Left-Most are “root” Functions, not Directly Called by C Code**

Call Graph (cont.)

- **Starred Functions use `auto`/ Parameter RAM that does not Overlap with Other Functions**
  - Look at these functions to reduce RAM usage

Call Graph (cont.)

```
Machine type is 18F452

Call graph:

*_main size 0,6 offset 0
*    _dummy size 0,5 offset 6
*        _fcp size 2,12 offset 11
         awtoft size 0,0 offset 11
     _free size 0,2 offset 6
*_another_isr size 0,0 offset 25
*_my_isr size 0,6 offset 25
     lbtoft size 0,0 offset 31
*    _delay size 2,0 offset 31
```

Psect Summary Details

- **Each psect Defined by Each Module Contributing to the Output is shown with:**
  - Psect link and load address
    - **Load address specifies ROM (HEX) image if applicable**
  - Psect length (size)
  - Resident memory space

Psect Summary (cont.)

```
Name           Link   Load  Length Select Space Scale

startup.obj
   end_init       A      A      4      0      0
   init           0      0      A      0      0
main.obj
   text        3FEA   3FEA     16   1FF5      0
   bss            6      6     10     FE      1
   my_bit         8      1      2      0      1     8
C:\Program Files\...\lib\pic84--p.lpp

COMMON
   param         1A     1A     10      0      1
```

Memory/Symbol Summary

- **Unused Space Remaining in Class is Indicated**
  - Classes defined by linker option
  - More than one class may cover an address range
- **Symbol Table for Global Symbols**
  - Shows assembler symbol name
  - Residing psect (or abs) and address

Memory Summary (cont.)

```
UNUSED ADDRESS RANGES

        BIGRAM                 000003-0005FF
        CODE                   000022-007FFF
        COMRAM                 000003-00007F
        CONFIG                 300000-30000D
        EEDATA                 F00000-F000FF
        IDLOC                  200000-200007
        RAM                    000003-0005FF

                     Symbol Table

__HRAM      (abs)   000000     __Hbigbss  bigbss 000003
__Hbigdata  bigdata 000003     __Hbss     bss    00000A
_i          bss     00000A     _main      text   000A06
```

Lab 1

- **Open lab1 Project in MPLAB® IDE**
- **Using .lst/.map File, determine:**
  - Address of the C function `get_half`
  - Address of the C variable `randx`
  - Unused space in `CODE` class
  - Did the function `delay` appear in the call graph?
    - **Did it appear in the output assembly code?**

Lab 1 (cont.)

- Were any modules linked in from the library files?
- Which functions had their auto/parameter areas overlapped with that from other functions?

- Compiler, HI-TIDE & C-Wiz
- Data Types & Qualifiers
- Diagnostic Files & Options
- **Interrupts**
- Library & Compiler-Generated Code
- Psects & the Linker
- HI-TECH Assembly

C Interrupt Functions

- **An Interrupt Function (ISR) is Defined by the Qualifier `interrupt`**
  - Associated with high-priority interrupt vector by default

```
void interrupt isr(void)
{
  if(RCIF && RCIE)
    byte = RCREG;
}
```

Interrupt Functions (cont.)

- Low-priority ISR can be created by also using the `low_priority` keyword

```
void low_priority interrupt
isr(void)
{
  if(T0IF && T0IE)
    count++;
}
```

Context Restoration

- **Different Memory Areas are used for the Context of the Low and High-Priority Interrupt Routines**
- **The Compiler Selectively Saves those Objects Used by the ISR**
  - Objects include registers used by the CGEN, and scratch variables
  - High-priority ISRs take advantage of shadow registers

Context Restoration (contd.)

- Compiler takes into account registers used by functions called by the ISR
- In-line assembler cannot be scanned for register usage

- **“Unseen” Routines Called by an ISR Forces Save of All Registers**
  - CGEN will see any called C routine defined above ISR in the module
  - CGEN will see any called C routine

Context restoration (contd.)

- **Interrupt Functions are not Re-entrant**
- **Functions Called from Interrupt Functions and Main-Line Code:**
  - Produce a linker error “Function appears in multiple call graphs…”
  - Produce duplicate assembly output for each call tree

Lab 2

- **Open lab2 Project in MPLAB® IDE**
- **Identify the ISR**
- **Which Variables:**
  - should be qualified `volatile`?
  - are **not** assigned by atomic operations?
- **Note the Assembly Code that Assigns to `bb`**
  - Make `bb` volatile & note change

Lab 2 (cont.)

- **What is the Name of the Library Routine Implicitly Called to Perform Division in `main`?**
  - Was the same routine called by the ISR?

- Compiler, HI-TIDE & C-Wiz
- Data Types & Qualifiers
- Diagnostic Files & Options
- Interrupts
- **Library & Compiler-Generated Code**
- Psects & the Linker
- HI-TECH Assembly

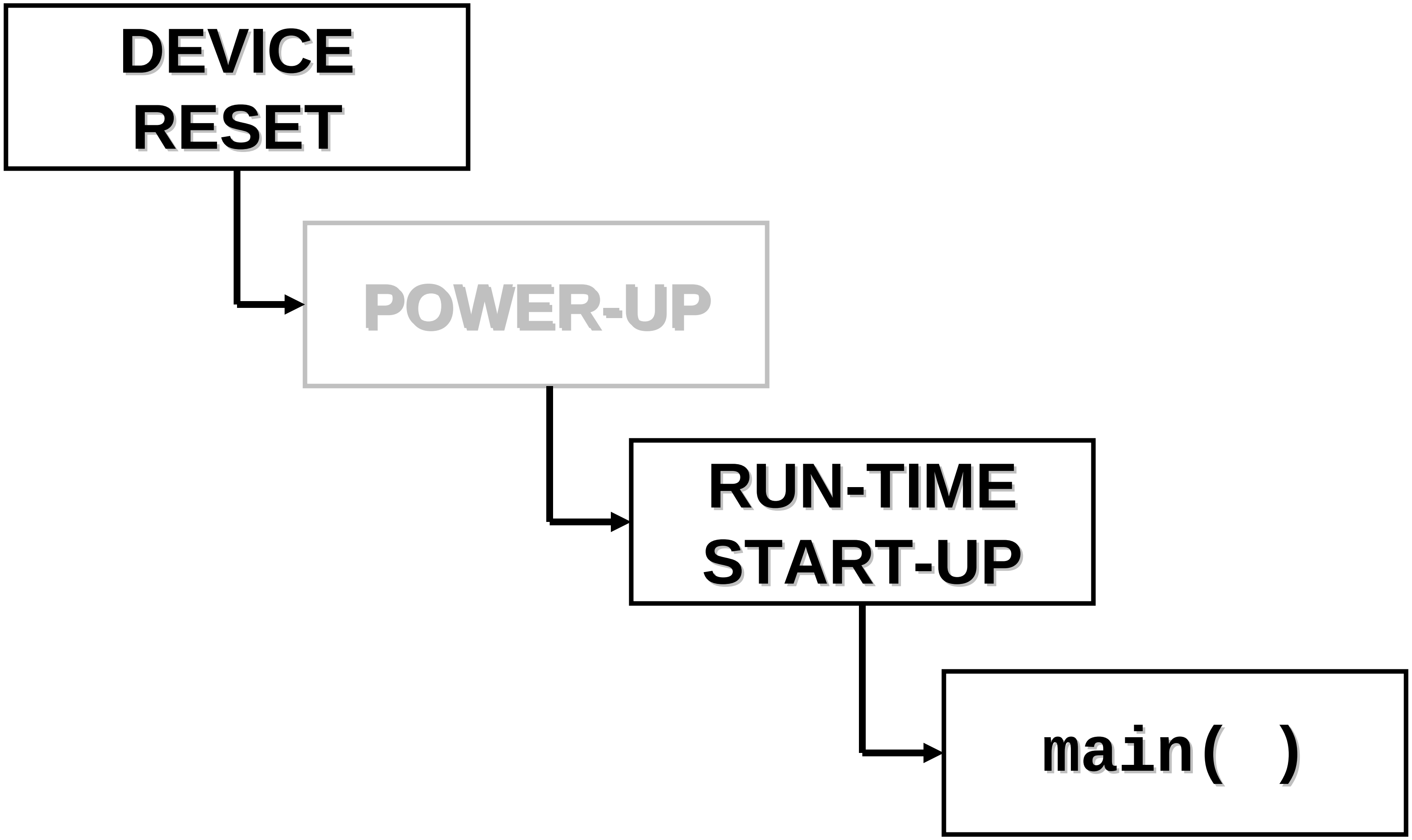
Library Code

- Libraries Functions (string, math etc.) Provided
- Driver Links Relevant Library Files
  - See map file to confirm
- Only Functions Used are Included
- Source Code is Searched First for Function Definitions
  - See map file to confirm

Compiler-Generated Code

- **Some Routines are Written or Customized by the Compiler:**
  - Run-time start-up code (assembly)
  - Printf (C code)

Coming out of Reset



Run-Time Start-up Code

- **The Run-Time Start-up Code:**
  - Runs any user's power-up code
  - Clears uninitialized variables
  - Assigns values to initialized variables
  - Performs any miscellaneous setup
  - Executes `main`
- **Code Contained in Assembly File: `startup.as`**

Run-Time Start-up (cont.)

- Can be Controlled by `–runtime` Suboptions:

`init`	initialization of variables
`clear`	clearing of variables

Power-up Routine

- **The Power-up Routine is Executed after Reset**
- **Its Use is Automatic provided:**
  - Code is within the `powerup` psect
  - It jumps to `start` on completion

`printf` Routine

- **Extra CGEN pass detects `printf` and placeholders used**
  - Symbols are defined which customize a generic `printf` routine
- **Options: `-Ll`, `-Lf` or `–Lw` must be used to select `printf` library version**

`printf` Routine (cont.)

- User must "define" stdout by Writing the `putch` Function

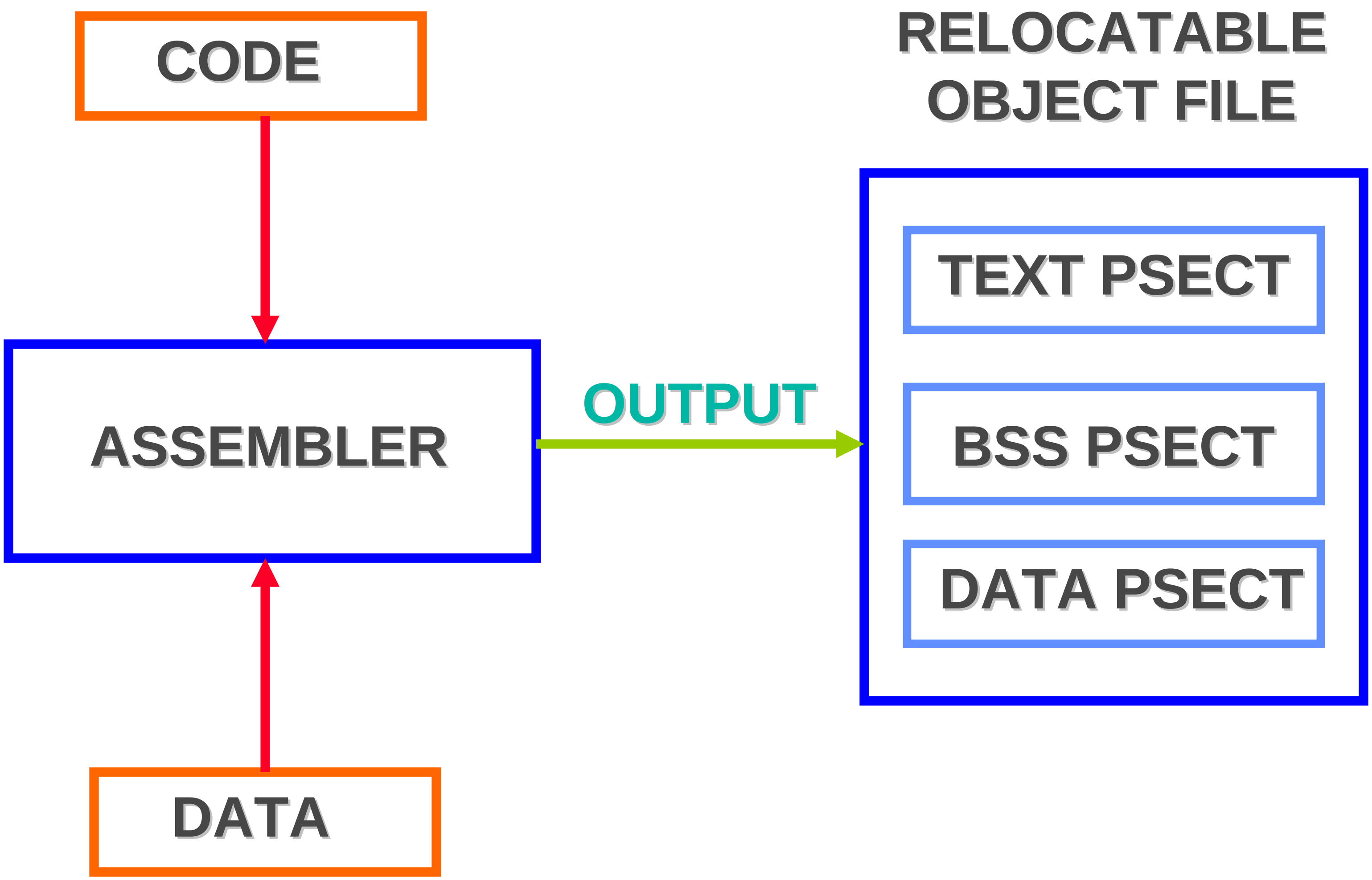
```
void putch(char data) {
  while( ! TRMT)
    ;
  TXREG = data;
}
```

Lab 3

- Open lab3 Project in MPLAB® IDE
- Add Option to Keep `startup.as`
- Compile Project and Inspect `startup.as`
- Add `powerup.as` File to Project
- Recompile and Re-examine `startup.as`
- Confirm Use in Map File

- Compiler, HI-TIDE & C-Wiz
- Data Types & Qualifiers
- Diagnostic Files & Options
- Interrupts
- Library & Compiler-Generated Code
- **Psects & the Linker**
- HI-TECH Assembly

PSECTs


CODE
ASSEMBLER
DATA
OUTPUT
RELOCATABLE
OBJECT FILE
TEXT PSECT
BSS PSECT
DATA PSECT

PSECTs (cont.)

- **The Code Generator places Output into a psect by Using an Assembler Directive (PSECT)**
  - The directive specifies the psect name and any options
- **The Assembler produces a Relocatable Object File that consists of psects**

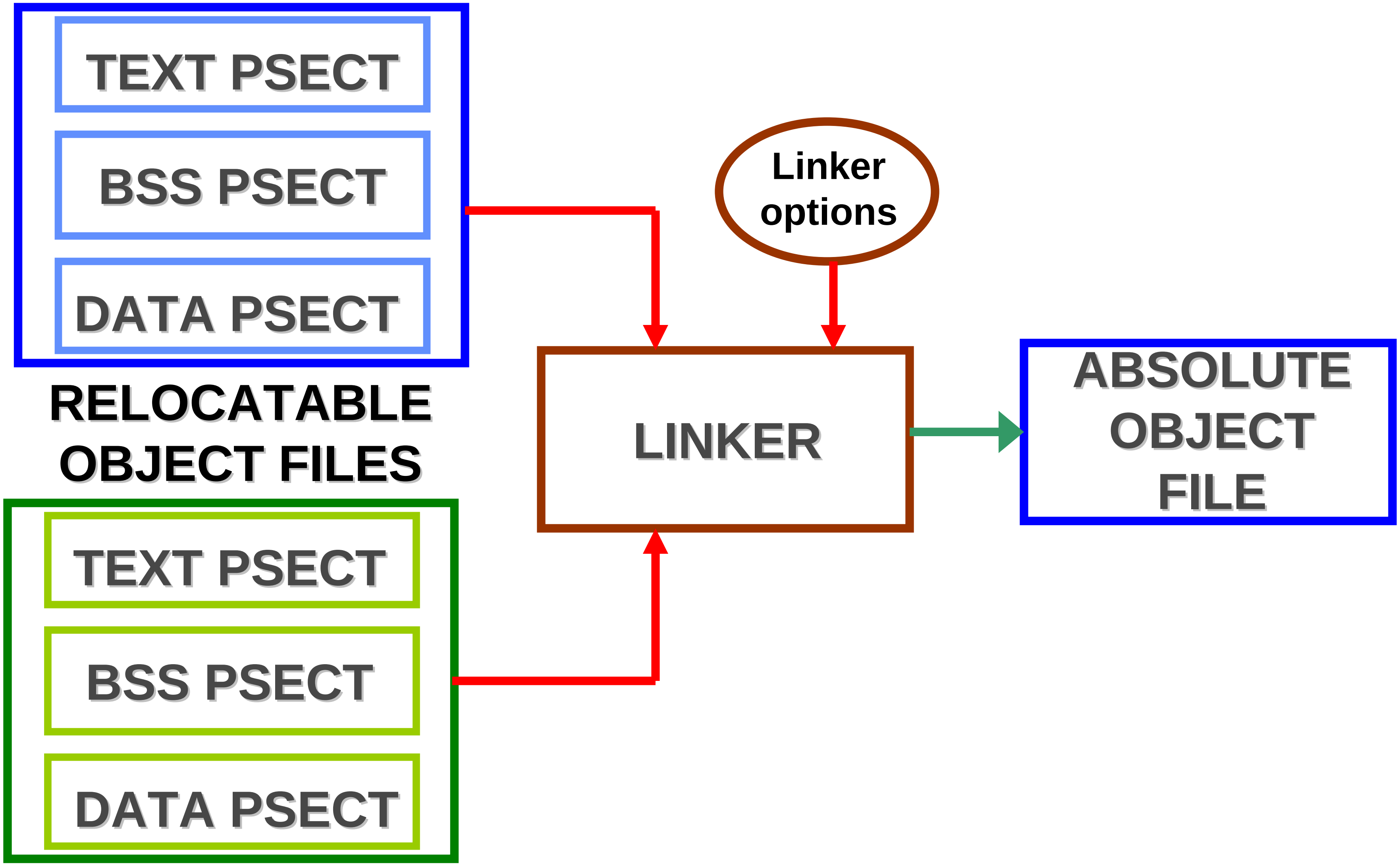
PSECTs (contd.)

- **Compiler-Generated psects, e.g.:**
  - `text` for code
  - `data` types for initialized variables
- **Additional psects can be Created by the Programmer**

PSECT Flags

global	group with other global psects
delta	specify size of addressing unit
class	make member of a linker class
reloc	specify psect alignment
space	specify memory space
ovlrd	overlay with similar psects
limit	specify upper address limit
bit	psect holds bit objects

Linker



TEXT PSECT
BSS PSECT
DATA PSECT
RELOCATABLE OBJECT FILES
TEXT PSECT
BSS PSECT
DATA PSECT
Linker options
LINKER
ABSOLUTE OBJECT FILE

Linker (cont.)

- **The Output of the Linker is an Absolute Object File**
- **The Linker performs Memory Allocation in Several Steps**
  - Grouping of psects by name, obeying any `psect` flags
  - Relocation of psects into memory as specified by linker options and `psect` flags

Linker (cont.)

- **Resolution of Symbolic Values**
  - Symbol fix-up to absolute addresses in object file
  - Rewrite (fix-up) of assembler list file

Linker Options

- **Psects may be Linked**
  - Explicitly in a set order and address
  - Anywhere within a class range

`-Ptext=200h`	**text at 200h**
`-Ptext=200h,const`	**const after text**
`-Pstrings=const`	**const after strings**

`-ACODE=0-7FFh,`
`800-FFFh`
	define CODE range

`-ptext=CODE`	**text anywhere in class**

Linker Options

- **Additional Linker Options can be Added Using the Driver -L- Option**
- **Default Linker Options can be Modified Using the Same Option**
  - If the new option string up to the first “=“ matches a default linker option, it replaces that default option

Linker Options (cont.)

Example: Given default linker options:

```
-pparam=100h,bss
```

place bss psect at 200h

```
-L-pparam=100h –L-pbss=200h
```

Question

What does this linker error mean?

```
fixup overflow in expression (location
0x302 (0x300+2), size 1, value 0x116)
```

```
000300   0E01    movlw   055h
000302   6F00’   movwf   _c
```

linker allocates _c at address 0116h

```
0110 1111 xxxx xxxx    movwf _c
        1 0001 0110
```

Question

What does this linker error mean?

```
Can't find space for 0xF40 words (0xF40
withtotal) for psect text in segment CODE
```

```
UNUSED ADDRESS RANGES
  CODE    3900-3FFF
          7650-7FFF
```

total free space 10AEh words in two blocks

Linker Options

- **CGEN output can be redirected into a new psect using:**

```
#pragma psect current=new
```

  - What was placed in the psect *`current`* will now be placed in *`new`*
  - Has effect over entire module

- Compiler, HI-TIDE & C-Wiz
- Data Types & Qualifiers
- Diagnostic Files & Options
- Interrupts
- Library & Compiler-Generated Code
- Psects & the Linker
- **HI-TECH Assembly**

Assembly

- **Assembly Code may be Written:**
  - In separate assembler modules; or
  - Placed in-line with C code using either:
    - **An `asm(“ … ”);` statement which places one instruction; or**
    - **An `#asm … #endasm` block of instructions**

Assembly

- **Be aware of the following:**
  - All assembly code may be altered by the assembler optimizer, if enabled
    - **Preserve code in a separate module compiled without the optimizer**
  - An `#asm` block is not syntactically part of the C code and may not follow normal C flow-of-control rules
  - Assembler code must not alter the state assumed by the CGEN

Common Assembler Directives

`PSECT`	Create & switch to psect
`GLOBAL`	Link with/make public symbols
`EQU`	Equate symbol and value
`DB, DW`	Place byte/word in psect
`DS`	Reserve space in psect
`ORG`	Move offset into psect
`SIGNAT`	Define signature for routine
`FNSIZE`	Specify space for autos/param

Lab 4

- **Open lab4 MPLAB® IDE Project**
- **Compile and Verify the Size of the Pointer Parameter to** check**, and the Code Generated for** check
  - Uncomment second assignment to pointer and repeat
  - Uncomment third assignment and repeat
- **Remove All Assignments to** c **and Observe the Code Produced for** check

UNIVERSITY OF MICROCHIP
UOFM
MASTERS 2007

Summary

- **Introduced:**
  - HI-TIDE & C-Wiz
  - PRO version compiler
  - New STD version features
- **Practical Use of List and Map Files**

University of Microchip UofM MASTERs 2007

Summary

- **You should now be able to:**
  - Control start-up code
  - Control object placement
  - Understand interrupt issues
  - Interact with assembly code

Tools Used in this Class

- MPLAB® IDE v7.61.00
- HI-TECH PICC-18 PRO v9.61
- HI-TIDE v3.12PL1

Thank You

Appendix

Optimization Techniques

University of Microchip UofM MASTERs 2007

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What’s Better
- PIC18 Considerations

PRO Version Optimizations

- **All C and p-code Library Modules are combined into one during the Code Generation Phase**
  - This allows the code generator to analyze more code
  - Better prediction of register usage in functions allows for better caching of variables in registers
  - Better optimization by assembler
  - The programmer does not need to rearrange functions between modules

PRO Version Optimizations (cont.)

- **Allocation of Objects into the Data Space is Automatic**
  - This ensures that the access bank is fully utilized
  - The programmer does not need to use keywords for variable placement

PRO Version Optimizations (cont.)

- **The Size of Pointers is determined from the Program, based on what the Pointer References and the Amount of Code and Data Defined**
  - This minimizes data and program memory usage
  - The programmer does not need to use options to control pointer size

PRO Version Optimizations (cont.)

- **The Required Size of Variables may be determined by its Content**
  - This minimizes data and program memory usage
  - The programmer does not need to modify code to get optimal output

PRO Version Optimizations (cont.)

- **The Code Required to Implement the printf Function can be determined by Placeholders used in the Format String**
  - This minimizes program memory usage
  - The programmer does not need to use options to specify which printf to use

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What’s Better
- PIC18 Considerations

HI-TECH PICC Compiler

- Basic Tips, it's the simplest of things:
  - **Turn the Optimizers on**
    - Versions 9.x and greater, turned on by default
  - **In MPLAB IDE, turn them on from the Build Options menu**
  - **Use the smallest data types that will do the job**
    - *unsigned char* for 8-bit values, unsigned *int* for 16-bit values
    - If a function only needs to return true or false, use *bit* value. If you are concerned with portability and the use of *bit*, call it a BOOL and use a define or typedef

HI-TECH PICC Compiler

- Basic Tips, it's the simplest of things:
  - **Use *unsigned* types rather than *signed*, if possible**
  - **Try to reduce the number of mixed types within an expression**
    - Although the compiler will handle all casting for you, this can be costly in terms of code size, particularly when there is conversion from signed types to a large type, or from integral to floating point or vice versa
  - Use the *const* qualifier for strings
    - **The *const* qualifier makes access to an object read-only**
    - **It tells the compiler that the object might be able to be stored in program memory rather than taking up RAM**

HI-TECH PICC Compiler

- Basic Tips, it’s the simplest of things:
  - **With function declarations, don't use too many parameters.**
    - Although you may not want to communicate using global variables, it can be appropriate to consider using one if it represents something global anyway, e.g. state machine context, ports or peripherals. Globals eliminate the need to copy the variable value each time a function is called, reducing code size and speeding up execution.
    - However, there are advantages and disadvantages with this sort of thing
  - **On the subject of using global variables:**
    - (1) if a function returns several things, putting them in global variables or a global struct may be more code and time-efficient than having the function return a struct (at the expense of having the items always allocated)
    - (2) if a function takes one or more ‘bit’ parameters, it may be worthwhile to create global variables for them and define wrapper macros

HI-TECH PICC Compiler

- Instead of:

```
typedef unsigned char ub;
void foo(ub ch, ub mode) /* Mode is always 0 or 1 */
```

- Use:

```
bit foo_mode;
#define foo(ch,mode)
    (foo_mode=(mode),do_foo(ch))
void do_foo(ub ch)
```

- Allows faster testing of ‘mode’ within the function and will also, in most cases, make function calls more efficient

Reducing Code Size

- **Use** `near` **Variables whenever possible**
  - When defining pointers that only point to **near** objects, qualify the pointer
- **Use** `auto` **Objects rather than Global or Static Objects**
  - These all reside in the same bank

“Virtual Stack” Overlay Model for Local Variables

- **HI-TECH PICC creates an automatic overlay model for local variables. This allows shared RAM usage within functions that are not part of the same call tree:**

```
void main(void){
      Function1(A, B);
      Function2(D, E);}
Function1(unsigned char a, unsigned char b)
            PORTB = a | b;
Function2(unsigned char d, unsigned char e)
            PORTB = d & e;
```

In this example, d and e will overlay a and b and will use only (2) bytes of RAM

Variables in the Same Bank (PIC16 only)

- When assigning variables, consider sequences which are common and assign those variables to the same bank using the Bank keyword
- This eliminates any unnecessary banking instructions within a function sequence

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What’s Better
- PIC18 Considerations

HI-TECH PICC Compiler

- Initializing global variables

```
unsigned int value = 5;
main()
{
}
```

  - **At start-up, initialized variables should be set up and uninitialized variables should be set to zero. These initializations are handled by a start-up routine which is included *if required*. On a Mid-Range PIC® MCU, the above will compile to 61 words of program memory.**

HI-TECH PICC Compiler

- **Now if we make a small change:**

```
unsigned int value;
main()
{
    value = 5;
}
```

- Compiles to 29 words of program memory.
  - **The initialization routine that was previously required to set *value* did not have to be linked in. Note, there is a fixed overhead to initialized values; when you have less than ~20 bytes, it is more code efficient to assign them as shown here.**

HI-TECH PICC Compiler

- **If you only have a small number of variables that need to be initialized, then you may find it better to do it in main()**
- **On the other hand, if you have lots of variables to initialize, then it may become more efficient to let the compiler set them up in one block at start-up**

The PRO Version compiler will *write* custom start-up code based on exactly what is needed

HI-TECH PICC Compiler

- Basic Tips, it's the simplest of things:
  - **Anything that is done multiple times can be moved into its own function**
    - This can be worthwhile, even if it is just one line containing a moderately complex expression. However, in doing is advantageous when there are very few parameters since parameter passing is also costly.
    - Parameters are stored on a virtual overlay RAM stack so memory is reused when functions don't call each other
    - If you have lots of parameters and only a few lines of code, a macro will likely have less overhead since the call and return are eliminated
  - **It can also be interesting to have a look at the assembly just to see if there are any lines that produce unexpectedly large amounts of code and if so, try doing something else.**

HI-TECH PICC Compiler

- Basic Tips, it’s the simplest of things:
  - **When initializing an array, use something other than a "for" loop and count downwards**
    - This is because the decrement and test for leaving the loop is one instruction

```
unsigned char somearray[10];
unsigned char n;
for (n = 0; n < 10; n++)
    somearray[n] = 0;   // commonly used but not ideal

n = 10;                 // This usually will produce smaller code
do
  somearray[n-1] = 0;   // the "-1" does not add any overhead since
while (--n);            // it is added to the address of somearray
```

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What's Better
- PIC18 Considerations

HI-TECH PICC Compiler

```
//example one
const unsigned char string1[] = "Hello, world!\n";
```

- Places the string “Hello, world!\n” into program memory
  - **Simple & clean**

HI-TECH PICC Compiler

```
//example two
const unsigned char * table[] = {
    { "one" },
    { "two" },
    { "three" },
};
```

- Creates an array of pointers to strings, ***bye-bye*** RAM
  - **Stores strings "one", "two", "three“ in program memory**
  - **There will also be a 16-bit pointer for each created in RAM**
  - **Additionally, the compiler will have to include code to initialize these pointers at start-up**

HI-TECH PICC Compiler

```
//example three
const unsigned char * const table[] = {
    { "one" },
    { "two" },
    { "three" },
};
```

- With "const" now being applied to the pointers, the pointers will be stored in program memory too
  - **Better than using RAM, but the problem with this type of declaration (called a ragged-array) is that pointers must be created**

HI-TECH PICC Compiler

```
//example four
const unsigned char table[3][6] = {
   { "one" },
   { "two" },
   { "three" },
};
```

- Now the array has defined dimensions
  - **This is only a small example and the difference is minimal, but when applied to a large array of strings it could make a big difference**

HI-TECH PICC Compiler

- (General): Rather than repeatedly accessing an element in an array in a sequence of code, it may be better to declare a pointer and set it to point at the element
  - **Then, only code for dereferencing the pointer needs to be generated instead of always having to first recalculate the address of the element**

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What's Better
- PIC18 Considerations

HI-TECH PICC Compiler

- Refer to the map file’s call graph to identify the calling relationships between functions
  - **Locate functions which make lots of calls and group the called functions into the same C file**

```
Call graph:
*_main size 0,0 offset 0
*    _caller size 0,0 offset 0
*        _func1 size 0,0 offset 0
*        _func2 size 0,0 offset 0
*        _func3 size 0,0 offset 0
```

  - ***main* calls *caller* which calls *func1, func2, func3***
  - **So, in the *very* simple example, it would be best to group func1, func2, func3 together in the same file as caller**

HI-TECH PICC Compiler

- Before:

  main.c: contains main()

  caller.c: contains caller()

  file1.c: contains func1()

  file2.c: contains func2()

  file3.c: contains func3()

- After:

  main.c: contains main()

  caller.c: contains caller(), func1, func2 and func3

HI-TECH PICC Compiler

- When functions are in the same file, the compiler can perform call/jump optimizations between functions (and possibly merging & other optimizations)
- When functions are in different files, these optimizations cannot occur because the location (and contents of) the other functions is not known
- The above would tend to indicate that it might be better to put your entire project into the one C file
  - **Well, if you are *really* struggling for space – then yes, this could be a good idea**
  - **However, from a design & maintenance point of view, it probably isn’t. Hi-Tech is currently developing a new code generator with whole program optimization capabilities which effectively does the above automatically. Look for this next year...**

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What’s Better
- PIC18 Considerations

HI-TECH PICC Compiler

- Case vs. If-Else, what's better??
  - **For example, in a simple 2 choice check/assignment routine…..**

```
if (a==0)
  b=3;
else if
(a==3)
  b=7;
```

```
switch (a)
{
case 0:
            b=3;
            break;
case 1:
            b=7;
            break;
default:
            break;
}
```

  - **What is a better choice?**

HI-TECH PICC Compiler

- What about, in a simple 15 choice check/assignment routine…..

```
if (a==0)
   b=0;
else if (a==1)
   b=1;
   .
   .
else if (a==14)
   b=14;
```

  - What is a better choice?

```
switch (a)
{
case 0:
            b=0;
            break;
case 1:
            b=1;
            break;
            .
            .
case 14:
            b=14;
            break;
default:
            break;
}
```

HI-TECH PICC Compiler

- Case **2 options, 26 words program memory (21) (17) (15)**
- If-Else **2 options, 22 words program memory (20) (12) (12)**

- Case **6 options, 58 words program memory (45) (38) (36)**
- If-Else **6 options, 66 words program memory (56) (37) (37)**

- Case **7 options, 66 words program memory (51) (43) (41)**
- If-Else **7 options, 77 words program memory (65) (43) (43)**

- Case **15 options, 74 words program memory (84) (74) (68)**
- If-Else **15 options, 165 words program memory (137) (91) (91)**

Note: (Opt.On lev 9) (Asm opt) (Asm opt & Opt.On lev 9)

HI-TECH PICC Compiler

- Case vs. If-Else
  - **When is one better than the other??**
  - **That depends. Per the previous example:**
    - If-Else for <=6 decisions
    - Case for >=6 decisions

HI-TECH Compiler Optimization

- Use PRO Version of the Compiler
- Some Simple things with Variables
- Initializing
- Strings & Things
- Functional Relationships
- If-Else vs. Case: What’s Better
- PIC18 Considerations

HI-TECH PICC-18 Compiler

- The *access bank* can be reached regardless of what bank is currently selected
  - Utilizing this area can reduce your code size because no bank swapping instructions will be required
  - The compiler offers the *near* storage qualifier to put an object into the access bank. The near qualifier can be used on any global or static variables
  - Ordinarily, pointers are 16 bits (or 24 bits) in size. A pointer qualified as near is only 8 bits in size which is much more efficient to work with.

HI-TECH PICC-18 Compiler

Near vars = Access bank examples

```
unsigned int number;              // no storage qualifier,
                                  // could be positioned anywhere

near unsigned int fastnum;        // will be put into the access bank

near unsigned int * fastptr;      // an 8-bit pointer that can point
                                  // to near int objects

near unsigned int * near fastptr2; // same as above, but the pointer
                                   // itself is also put into the access bank.
```

HI-TECH PICC-18 Compiler

- If a function needs to return multiple values, do this with a structure instead of with pointer parameters since pointers to non-near objects are costly

```
void GetTime (unsigned char* hour, unsigned char*
    minute, unsigned char* weekday);
....
unsigned char hour;
unsigned char minute;
unsigned char weekday;
GetTime (&hour, &minute, &weekday);
```

- Very inefficient…

HI-TECH PICC-18 Compiler

- Much better way…

```
typedef struct {
  unsigned char hour;
  unsigned char minute;
  unsigned char weekday;
} tTime;
tTime GetTime (void);
....
tTime Time = GetTime();
```

Common Math Libraries

- Review each of the math libraries called out in the MAP file
- If using 16/8 divide in one place and a 16/16 divide in another, promote the 8-bit variable to 16 bits to eliminate the 16/8 library function

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeeLoq, KeeLoq logo, microID, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AmpLab, FilterLab, Linear Active Thermistor, Migratable Memory, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, PICkit, PICDEM, PICDEM.net, PICLAB, PICtail, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Smart Serial, SmartTel, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.