SCX Reference Manual

Custom Computer Services, Inc.
September 2002

Table of Contents

OVERVIEW

PCW/SX C Overview

This compiler is specially designed to meet the special needs of the Scenix controllers. These tools allow developers to quickly design application software for these controllers in a highly readable high-level language.

The compiler has some limitations when compared to a more traditional C compiler. The hardware limitations make many traditional C compilers ineffective. As an example of the limitations, the compiler will not permit pointers to constant arrays. This is due to the separate code/data segments in the SX hardware. On the other hand, the compilers have knowledge about the hardware limitations and does the work of deciding how to best implement your algorithms. The compiler can implement very efficiently normal C constructs, as well as input/output operations and bit twiddling operations.

Technical Support

The latest software can be downloaded via the Internet at:

http://ccsinfo.com/download.shtml

for 30 days after the initial purchase. For one year's worth of updates, you can purchase a Maintenance Plan directly from CCS. Also found on our web page are known bugs, the latest version of the software, and other news about the compiler.

We strive to ensure that each upgrade provides greater ease of use along with minimal, if any, problems. However, this is not always possible. To ensure that all problems that you encounter are corrected in a diligent manner, we suggest that you email us at support@ccsinfo.com outlining your specific problem along with an attachment of your file. This will ensure that solutions can be suggested to correct any problem(s) that may arise. We try to respond in a timely manner and take pride in our technical support.

Secondly, if we are unable to solve your problem by email, feel free to telephone us at (262) 797-0455 x 32. Please have all your supporting documentation on-hand so that your questions can be answered in an efficient manner. Again, we will make every attempt to solve any problem(s) that you may have. Suggestions for improving our software are always welcome and appreciated.

Installation

Insert the CD-ROM and Run:

setup.exe

Invoking the DOS Compiler

To start the DOS compiler, enter SXC at the DOS prompt. The command may be optionally followed by a filename of a file to open. Multiple files may be specified separated by spaces. To compile a file without the IDE, place a -C before the filename.

File Formats

The compiler can output 8 bit hex, 16 bit hex, and binary files. Two listing formats are available. Standard format may be required by some third-party tools. The simple format is easier to read. The debug file may either be a .COD file, Advanced Transdata .MAP file or IEEE-695 file. All file formats and extensions are selected via the **options|file** formats window.

Direct Device Programming

The IDE has a program option in the main menu bar. When invoked, the IDE will issue a command to start the user's device programmer. The commands are specified in the Options|Program window. The %H is replaced with the HEX filename and %D is replaced with the device number. Put a ! at the end if the command line if you would like a pause before returning to IDE. Only programs that can be invoked by a command will work with this option.

Menu Commands

The integrated development environment allows edit windows to be opened, closed and re-arranged easily. When a compile command is issued the currently active window is compiled. Be aware all files are saved to disk when a compile command is issued.

Compile errors appear in the main window (in red) and the area in error will be highlighted.

In addition to opening any file, hot keys may be used to open windows with the memory map, assembly code listing and calling tree.

When exiting the software a prompt will remind you of any editor buffers that have not been saved.

Many WordStar keys are also accepted

Special Features

ASM/TREE/MAP/STATS (speed buttons)
After compiling, these buttons bring up various informational windows. A program must be compiled before these windows may be opened.

PROGRAM CHIP (speed button)
This button will invoke your device programmer software as specified in the Compile\Options window.

DISASSEMBLE (File|Disassemble)
This command will take a HEX file and generate an assembly file so that selected sections can be extracted and inserted into your C programs as inline assembly. Options will allow the selection of the assembly format.

STATUS LINE
Click on the left hand side of the status line to GOTO a specific line number.

Editor

Cursor Movement	
Left Arrow	Move cursor one character to the left
Right Arrow	Move cursor one character to the right
Up Arrow	Move cursor one line up
Down Arrow	Move cursor one line down
Ctrl Left Arrow	Move cursor one word to the left
Ctrl Right Arrow	Move cursor one word to the right
Home	Move cursor to start of line
End	Move cursor to end of line
Ctrl PgUp	Move cursor to top of window
Ctrl PgDn	Move cursor to bottom of window
PgUp	Move cursor to previous page
PgDn	Move cursor to next page
Ctrl Home	Move cursor to beginning of file
Ctrl End	Move cursor to end of file
Ctrl S	Move cursor one character to the left
Ctrl D	Move cursor one character to the right
Ctrl E	Move cursor one line up
Ctrl X	** Move cursor one line down
Ctrl A	Move cursor one word to the left
Ctrl F	Move cursor one word to the right
Ctrl Q S	Move cursor to top of window
Ctrl Q D	Move cursor to bottom of window
Ctrl R	Move cursor to beginning of file
Ctrl C	* Move cursor to end of file
Shift ~	Where ~ is any of the above: Extend selected area as cursor moves

Editor Commands

Ctrl-O	Open file
Ctrl-S	Save file
Shift F11	Close file
F12	Bring up help index
Shift F12	Bring up editor help
F1	Find a help topic matching keyword at cursor

Compile Options

Debug file options

COD	Standard debug file
RICE16 MAP	Used only be older RICE16 S/W
IEEE-695	Industry Standard File

Error file options

Standard	Current standard
Original	Older standard

Object file options

8 bit HEX	8 Bit Intel HEX file
16 bit HEX	16 bit Intel HEX file
Binary	Straight binary (No fuse info)

List format options

Simple	A basic format with C code and ASM
Standard	The MPASM standard format with machine code
Old	Older MPASM format

Object file extension	The file extension for a HEX file
List file extension	The file extension for a list file

Project Wizard

This function will allow you to easily create the backbone code for an application. The wizard will autogenerate program code for virtually every feature on a Scenix SX.

Disassemble

This command takes a binary file for input, such as a hex file. Using that, it will generate an assemble code in that output file.

File Menu

New	Creates a new file
Open	Opens a file into the editor. If there are no other files open then the project name is set to this files name.
Save	Saves the file currently selected for editing.
Save As	Prompts for a filename to save the currently selected file.
Close	Closes the file currently open for editing. Note that while a file is open in PCW/SXC for editing no other program may access the file.
Close All	Closes all files.
Print	Prints the currently selected file.
Exit	Terminates PCW/SXC

Project Menu

New	Creates a new project. A project may be created manually or via a wizard. If created manually only a .PJT file is created to hold basic project information. An existing .C main file may be specified or a empty one may be created. The wizard will allow the user to specify project parameters and when complete a .C, .H and .PJT file are created. Standard source code and constants are generated based on the specified project parameters.
Open	A .PJT file is specified and the main source file is loaded.
Open All Files	A .PJT file is specified and all files used in the project are opened. In order for this function to work the program must have been compiled in order for the include files to become known.
Print All Files	All files in the project are printed. In order for this function to work the program must have been compiled in order for the include files to become known.
Include Dirs	Allows the specification of each directory to be used to search for include files for just this project. This information is saved in the .PJT file.

Edit Menu

Undo	Undoes the last deletion.
Cut	Moves the selected text from the file to the clipboard.
Copy	Copies the selected text to the clipboard.
Paste	Copies the clipboard contents to the cursor location.
Copy from file	Copies the contents of a file to the cursor location.
Copy to file	Copies the selected text to a file.
Find	Searches for a specified string in the file.
Replace	Replaces a specified string with a new string.
Next	Performs another Find or Replace.
Find matching } or)	The text will be highlighted up to the corresponding } or).
Set bookmark	Sets a bookmark (0-9) at the cursor location.
Goto bookmark	Move the cursor to the specified bookmark (0-9).
Next Window	Sets the focus to the next tabbed file.
Prev Window	Sets the focus to the previous tabbed file

Options Menu

Real tabs	When selected the editor inserts a tab character (ASCII 9) when the TAB key is pressed. When it is not selected and the TAB key is pressed spaces are inserted up to the next tab position.
Tab size	Determines the number of characters between tab positions.
Auto indent	When selected and the ENTER is pressed the cursor moves to the next line under the first character in the previous line. When not selected the ENTER always moves to the beginning of the next line.
WordStar keys	When selected the editing keys are WordStar style. See EDITOR for more information.
Recall open files	When selected PCW/SXC will always start with the same files open as were open when it last shut down. When not selected PCW/SXC always starts with no files open.
File Formats	Allows selection of the output file formats.
Programmer options	Allows the specification of the device programmer to be used when the PROGRAM CHIP tool is selected.
Include Dirs	Allows the specification of each directory to be used to search for include files by default for newly created projects. This has no effect on projects already created (use Project|Include Dirs to change those).

Editor Options

WordStar commands will enable additional keystrokes recognized by the editors. See editor keys for more information.

Tabs allow you to set the number of space equated by a tab and whether or not the tabs are converted to spaces or left as tabs.

PCW/SXC New Project

This command will bring up a number of fill-in-the-blank forms about your new project. RS232 I/O and I2C characteristics, timer options, interrupts used, A/D options, drivers needed and pin names all may be specified in the forms. When drivers are selected, required pins will be selected by the tool and pins that can be combined will be. Final pins selections may be edited by the user. After all selections are made an initial .c and .h files are created with #defines, #includes and initialization commands require for your project. This is a fast way to start a new project. Once the files are created you can not return to the menus to make further changes.

Make Main File

The compiler keeps track of your main file (in lower right) and when a compile is done this file is compiled. The file is usually the first you open in the editor. To change the main file to the one you are currently editing, simply click on this option.

Compile Menu

Compiles the current project (name is in lower right) using the current compiler (name is on the toolbar).

View Menu

C/ASM	Opens the listing file in the read only mode. The file must have been compiled to view the list file. If open this file will be updated after each compile. The listing file shows each C source line and the associated assembly code generated for the line.
Symbol Map	Opens the symbol file in the read only mode. The file must have been compiled to view the symbol file. If open this file will be updated after each compile. The symbol map shows each register location and what program variables are saved in each location.
Call Tree	Opens the tree file in the read only mode. The file must have been compiled to view the tree file. If open this file will be updated after each compile. ALT-T- the call tree will show the structure of the program. A (inline) will appear after inline procedures that begin with @. After the procedure name is a number of the form s/n where s is the number of the procedure and n is the number of locations of code storage required. If S is? then this was the last procedure attempted when the compiler ran out of ROM space. RAM=xx indicated that the total RAM required for the function.
Statistics	Opens the stats file in the read only mode. The file must have been compiled to view the stats file. If open this file will be updated after each compile. The statistics file shows each function, the ROM and RAM usage by file, segment and name.
Binary file	Opens a binary file in the read only mode. The file is shown in HEX and ASCII.
COD Debug file	Opens a debug file in the read only mode. The file is shown in an interpreted form.

Tools Menu

Device Editor	This tool allows the essential characteristics for each supported processor to be specified. This tool edits a database used by the compiler to control the compilation. CCS maintains this database (Devices.dat) however users may want to add new devices or change the entries for a device for a special application. Be aware if the database is changed and then the software is updated the changes will be lost. Save your DEVICES.DAT file during an update to prevent this.

Device selector	This tool uses the device database to allow a parametric selection of devices. By selecting key characteristics the tool displays all eligible devices.
View Data Sheet	This tool will bring up Acrobat Reader with the manufacture data sheet for the selected part. If data sheets were not copied to disk then the CCS CD ROM or a manufacture CD ROM must be inserted.
Numeric Converter	A conversion tool to convert between decimal, hex and float.
Serial Port Monitor	An easy to use tool to connect to a serial port. This tool is convenient to communicate with a target program over an RS232 link. Data is shown in as ASCII characters and as raw hex.
Disassembler	This tool will take as input a HEX file and will output ASM. The ASM may be in a form that can be used as inline ASM.
Program Chip	This simply invokes device programmer software with the output file.
Internet	These options invoke your WWW browser with the requested CCS Internet page:

- **View recent changes**- Shows version numbers and changes for the last couple of months.
- **e-mail technical support**- Starts your e-mail program with CCS technical support as the To: address.
- **Download updates** Goes to the CCS download page. Be sure to have your reference number ready.
- **Data Sheets** A list of various manufacture data sheets for devices CCS has device drivers for (such as EEPROMs, A/D converters, RTC...)

Help Menu

About	Shows the version of the IDE and each installed compiler.
Contents	The help file table of contents.
Index	The help file index.
Keyword at cursor	Does a index search for the keyword at the cursor location. Press F1 to use this feature.

Program

This command will invoke the device programmer software of your choice. Use the compile options to establish the command line.

Utility Programs

SIOW

SIOW is a simple "dumb terminal" program that may be run on a PC to perform input and output over a serial port. SIOW is handy since it will show all incoming characters.

DEVEDIT

DEVEDIT is a Windows utility that will edit the device database. The compiler uses the device database to determine specific device characteristics at compile time. This utility will allow devices to be added, modified or removed. To add a device, highlight the closest equivalent chip and click on ADD. To edit or delete, highlight the device and click on the appropriate button.

PCONVERT

PCONVERT is a Windows utility that will perform conversions from various data types to other types. For example, Floating Point decimal to 4 BYTE Hex. The utility opens a small window to perform the conversions. This window can remain active during a PCW/SCX. This can be useful during debugging.

Program Syntax

A program is made up of the following four elements in a file. These are covered in more detail in the following paragraphs.

Comment
Pre-Processor Directive
Data Definition
Function Definition

Comment

A comment may appear anywhere within a file except within a quoted string. Characters between the /* and */ are ignored. Characters after a // up to the end of a line are also ignored.

PRE-PROCESSOR COMMAND SUMMARY

Pre-Processor Command Summary	
Standard C	**Device Specification**
#DEFINE ID STRING	#DEVICE CHIP
#ELSE	#ID NUMBER
#ENDIF	#ID "filename"
#ERROR	#ID CHECKSUM
#IF expr	#FUSES options
#IFDEF id	**Built-in Libraries**
#INCLUDE "FILENAME"	#USE DELAY CLOCK
#INCLUDE <FILENAME>	#USE FAST_IO
#LIST	#USE FIXED_IO
#NOLIST	#USE I2C
#PRAGMA cmd	#USE RS232
#UNDEF id	#USE STANDARD_IO
Function Qualifier	**Memory Control**
#INLINE	#ASM
#INT_GLOBAL	#BIT id=const.const
#INT_xxx	#BIT id=id.const
#SEPARATE	#BYTE id=const
Pre-Defined Identifier	#ENDASM
_ _ DATE_ _	#RESERVE
_ _ DEVICE_ _	#ROM
	#ZERO_RAM
	Compiler Control
	#CASE
	#OPT n
	#PRIORITY
	#ORG

Pre-Processor Directives

Pre-processor directives all begin with a # and are followed by a specific command. Syntax is dependent on the command. Many commands do not allow other syntactical elements on the remainder of the line. A table of commands and a description is listed on the previous page.

Several of the pre-processor directives are extensions to standard C. C provides a pre-processor directive that compilers will accept and ignore or act upon the following data. This implementation will allow any pre-processor directives to begin with #PRAGMA. To be compatible with other compilers, this may be used before non-standard features.
Example: (Both of the following are valid)

```
#INLINE
#PRAGMA INLINE
```

#ASM
#ENDASM

The lines between the #ASM and #ENDASM are treated as assembly code to be inserted. These may be used anywhere an expression is allowed. The syntax is described on the following page. The predefined variable _RETURN_ may be used to assign a return value to a function from the assembly code. Be aware that any C code after the #ENDASM and before the end of the function may corrupt the value.
Example:

```
int find_parity (int data)      {
int count;
          mov           w,#0x8
          mov           count,w
          mov           w,#0
loop:
          xor           w,data
          rr            data
          decsz         count
          jmp           loop
          mov           return,w
}
```

#ASM COMMAND SUMMARY

Logical Operations	**Data Movement Instructions**
AND fr, W	MOV fr, W
AND W, fr	MOV W, fr
AND W, #lit	MOV W, fr-W
NOT fr	MOV W,#lit
OR fr, W	MOV W, /fr
OR W, fr	MOV W, --fr
OR W, #lit	MOV W, ++fr
XOR fr, W	MOV W, <<fr
XOR W, fr	MOV W, >>fr
XOR W, #lit	MOV W, <>fr
	MOV W, M
Arithmetic and Shift Operations	MOVSZ W,--fr
ADD fr, W	MOVSZ W, ++fr
ADD W, fr	MOV M, W
CLR fr	MOV M, #lit
CLR W	MOV !rx,W
CLR !WDT	MOV !OPTION, W
DEC fr	TEST fr
DECSZ fr	
INC fr	**Program Control Instructions**
INCSZ fr	CALL addr8
RL fr	JMP addr9
RR fr	NOP
SUB fr, W	RET
SWAP fr	RETP
	RETI
Bitwise Operations	RETIW
CLRB fr, bit	RETIW
SB fr.bit	RETW lit
SETB fr.bit	
SNB fr.bit	**System Control Instructions**
	BANK addr12
	IREAD
	PAGE addr12
	SLEEP

#BIT

This directive will create an identifier "id" that may be used just as any SHORT INT (one bit). The identifier will reference an object at memory location x with the bit-offset y. x may be a constant or another identifier. y must be a constant.

Examples:

```
#bit time_out = 3.4
int result;
#bit result_odd = result.0
...
if (result_odd)
...
```

#BYTE

This directive will create an identifier "id" that may be used just as any INT (one byte). The identifier will reference an object at memory location x. x may be a constant or another identifier.

If x is another identifier then "id" will be located at the same address as the other identifier. If the "id" already exists then that id will be located at the specified address.

Examples:

```
#byte  status = 3
#byte  b_port = 6
struct  {
  short int r_w;
  short int c_d;
  int unused : 2;
  int data    : 4; } a_port;
#byte a_port = 5
...
a_port.c_d = 1;
```

#CASE

Will cause the compiler to be case sensitive. By default the compiler is case insensitive.

_ _ DATE_ _

This preprocessor identifier contains the current date (at compile time) in the form: "30-DEC-99"
Example:

```
printf("Software was compiled on ");
printf(__DATE__);
```

#DEFINE

Used to provide a simple replacement of the ID with the given string. The following are some examples:
Example:

```
#define  BITS  8
#define hi(x)  (x<<4)
a=a+BITS;   //same as   a=a+8;
a=hi(a);    //same as   a=(a<<4);
```

#DEVICE

This directive defines to the compiler what the hardware architecture is. This will determine the RAM and ROM map as well as the instruction set used. You may select either 8 bit or 5 bit pointers. To use 5 bit pointers add a *=5 after the chip name or by itself after the chip is defined.

Examples:

```
#device SX20AC
#device *=5
```

## _ _DEVICE_ _

This pre-processor identifier is defined by the compiler with the base number of the current device (from a #device). The base number is usually the number after the C in the part number. For example the SX28AC has a base number of 28.
Example:

```
#if __device__==28
setup_port_a( ALL_DIGITAL );
#endif
```

#ERROR

This directive will cause the compiler to stop and issue an error with the message given after the #error. The message may include macros that will be expanded for the display. This may be used to see the macro expansion. The command may also be used to alert the user to an invalid compile time situation.

Examples:

```
#if  BUFFER_SIZE>16
#error  Buffer size is too large
#endif
#error   Macro test:  min(x,y)
```

#FUSES

This directive defines what fuses should be set in the part when it is programmed. This directive does not affect the compilation; however, the information is put in the output files. If the fuses need to be in Parallax format, add a PAR option. SWAP has the special function of swapping the high and low BYTES of non-program data in the Hex file. This is required for some device programmers.

Some common options are:

- LP, XT, HS, RC
- INT4MHZ INT2MHZ
- INT1MHZ INT500KHZ
- INT250KHZ INT125KHZ
- INT62KHZ INT31KHZ
- WDT NOWDT
- SYNC NOSYNC
- PROTECT NOPROTECT

Example:

```
#fuses  HS,WDT
```

#ID Number
#ID number, number, number, number
#ID Checksum

This directive defines the ID word to be programmed into the part. This directive does not affect the compilation but the information is put in the output file.

The first syntax will take a 16-bit number and put one nibble in each of the four ID words in the traditional manner. The second syntax specifies the exact value to be used in each of the four ID words.

When a filename is specified the ID is read from the file. The format must be simple text with a CR/LF at the end. The keyword CHECKSUM indicates the device checksum should be saved as the ID.

Examples:

```
#id  0x1234
#id  "serial.num"
#id  CHECKSUM
```

#IF
#ELSE
#ENDIF

The pre-processor evaluates the constant expression and if it is non-zero will process the lines up to the optional #ELSE or the #ENDIF.

Example:

```
#if  (a_size+b_size+c_size)>8
   printf("debug point a");
#else
printf("debug point b");
#endif
```

#ELSE
#ENDIF

The pre-processor evaluates the constant expression and if it is non-zero will process the lines up to the optional #ELSE or the #ENDIF.

Example:

```
#if  (a_size+b_size+c_size)>8
   printf("debug point a");
#else
printf("debug point b");
#endif
```

#IFDEF
#IFNDEF

This directive acts much like the #IF except that the preprocessor simply checks to see if the specified ID is known to the preprocessor (created with a #DEFINE). #IFDEF checks to see if defined and #IFNDEF checks to see if it is not defined.

Example:

```
#ifdef  DEBUG
   printf("debug point a");
#else
printf("debug point b");
#endif
```

#INCLUDE
#INCLUDE <filename>
#INCLUDE "filename"

Text from the specified file is used at this point of the compilation. The <> causes the compiler to look in the system directories file for the file.

Example:

```
#include  <SX28AC.H>
```

The ""causes the compiler to look in the current directory for the file.

Example:

```
#include "common.h"
```

#INLINE

This directive tells the compiler that the procedure immediately following the directive is to be implemented INLINE. This will cause a duplicate copy of the code to be placed everywhere the procedure is called. This is useful to save stack space and to increase speed. Without this directive the compiler will decide when it is best to make procedures INLINE.

Example:

```
#inline
swapbyte(int &a, int &b) {
   int t;
   t=a;
   a=b;
   b=t;
}
```

#INT

INT_RB0
INT_RB1
INT_RB2
INT_RB3
INT_RB4
INT_RB5

INT_RB6
INT_RB7
INT_RB
INT_RTCC
INT_GLOBAL
INT_TIMER1
INT_T1_CAPT
INT_T1_COMP
INT_TIMER2
INT_T2_CAPT
INT_T2_COMP

These directives specify the following function is an interrupt function. Interrupt functions may not have any parameters. Not all directives may be used with all parts.

The compiler will generate code to jump to the function when the interrupt is detected. It will generate code to save and restore the machine state, and will clear the interrupt flag. See SX_STWT.C for a full example program.

Example:

```
#int_rtcc
rtcc_handler() {
    overflow=true;
}
```

#INT_GLOBAL

This directive causes the following function to replace the compiler interrupt dispatcher. The function is normally not required and should be used with great caution. When used, the compiler does not generate start-up code or clean-up code, and does not save the registers.

#LIST

#List begins inserting or resumes inserting source lines into the .LST file after a #NOLIST.

#NOLIST

Stops inserting source lines into the .LST file (until a #LIST).

#OPT

This directive is only used with the PCW/SCX package. The optimization level is set with this directive. The directive applies to the entire program and may appear anywhere in the file. Optimization level 5 will set the level to be the same as the DOS compilers. The PCW/SCX default is 9 for full optimization.

#ORG

This directive will fix the following function or constant declaration into a specialized ROM area.

Example:

```
#ORG 0x1F00, 0x1FF0
Void loader (){
.
.
.
}
```

#PRAGMA

This directive is used to maintain compatibility between C compilers. This compiler will accept this directive before any other pre-processor command. In no case does this compiler require this directive.

Example:

```
#pragma device  SX28AC
```

#PRIORITY

The priority directive may be used to set the interrupt priority. The highest priority items are first.

Example:

```
#priority rtcc,rb
```

#RESERVE

This directive allows RAM locations to be reserved from use by the compiler. #RESERVE must appear after the #DEVICE otherwise it will have no effect.

Example:

```
#DEVICE …
#RESERVE  0x7d, 0x7e, 0x7f
```

#ROM

The #ROM will allow the insertion of data into the .HEX file as shown in the following example:

Example:

```
#rom  0x2100={1,2,3,4,5,6,7,8}
```

#SEPARATE

This directive tells the compiler that the procedure IMMEDIATELY following the directive is to be implemented SEPARATELY. This is useful to prevent the compiler from automatically making a procedure INLINE. This will save ROM space but it does use more stack space. The compiler will make all procedures marked SEPARATE, separate, as requested, even if there is not enough stack space to execute.

Example:

```
#separate
swapbyte (int *a, int *b) {
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

#UNDEF

The specified pre-processor ID will no longer have meaning to the pre-processor.

#USE

#USE DELAY

This directive tells the compiler the speed of the processor and enables the use of the built-in functions: DELAY_MS and DELAY_US. Speed is in cycles per second. An optional restart_WDT may be used to cause the compiler to restart the WDT while delaying.

Example:

```
#use delay (clock=20000000)
#use delay (clock=32000, RESTART_WDT)
```

#USE FAST_IO

This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another #USE directive is encountered. The fast method of doing I/O will cause the compiler to perform I/O without programming of the direction register. Port may be A-G.

Example:

```
#use fast_io(A)
```

#USE FIXED_IO

This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another #USE directive is encountered. The fixed method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. The pins are programmed according to the information in this directive (not the operations actually performed). This saves the byte of RAM used in standard I/O.

Example:

```
#use fixed_io(a_outputs=PIN_A2 ,PIN_A3)
```

#USE I2C

The I2C library contains functions to implement an I2C bus. The #USE I2C remains in effect for the I2C_START, I2C_STOP, I2C_READ, I2C_WRITE and I2C_POLL functions until another USE I2C is encountered. Software functions are generated unless the NOFORCE_SW is specified. The SLAVE mode should only be used with the built-in SSP.

Options:

• **MASTER**	Set the master mode
• **SLAVE**	Set the slave mode
• **SCL=pin**	Specifies the SCL pin (pin is a bit address)
• **SDA=pin**	Specifies the SDA pin
• **ADDRESS=nn**	Specifies the slave mode address
• **FAST**	Use the fast I2C specification
• **SLOW**	Use the slow I2C specification
• **RESTART_WDT**	Restart the WDT while waiting in I2C_READ
• **NOFORCE_SW**	Use hardware I2C functions.

Examples:

```
#use I2C(master, sda=PIN_B0,
    scl=PIN_B1)
#use I2C(slave,sda=PIN_C4,scl=PIN_C3
    address=0xa0,NOFORCE_SW)
```

#USE RS232

This directive tells the compiler the baud rate and pins used for serial I/O. This directive takes effect until another RS232 directive is encountered. The USE DELAY directive must appear before this directive can be used. This directive enables use of built-in functions such as GETCH, PUTCHAR, and PRINTF. Be sure to specify a FIXED_IO or FAST_IO directive before this directive if the I/O is *not* STANDARD_IO.

OPTIONS:

• RESTART_WDT	Will cause GETC() to clear the WDT as it waits for a character.
• INVERT	Invert the polarity of the serial pins (normally not needed when level converter, such as the AX232). May not be used with the internal SCI.
• PARITY=X	Where x is N, E, or O.
• BITS =X	Where x is 5-9 (5-7 may not be used with the SCI).

• FLOAT_HIGH	The line is not driven high. This is used for open collector outputs.
• ERRORS	Used to cause the compiler to keep receive errors in the variable RS232_ERRORS and to reset errors when they occur.
• FLOAT_HIGH	The line is not driven high. This is used for open collector outputs.
• ENABLE=pin	The specified pin will be high during transmit. This may be used to enable 485 transmit.

Example:

```
#use rs232(baud=9600, xmit=PIN_A2,rcv=PIN_A3)
```

The definition of the RS232_ERRORS is as follows:

- Bit 7 is 9th bit for 9 bit data mode (get and put).
- Bit 6 set to one indicates a put failed in float high mode.
- Bit 0 indicates a parity error.

#USE STANDARD_IO

This directive affects how the compiler will generate code for input and output instructions that follow. This directive takes effect until another #USE directive is encountered. The standard method of doing I/O will cause the compiler to generate code to make an I/O pin either input or output every time it is used. This requires one byte of RAM for every port set to standard I/O. Port may be A-E.

Example:

```
#use standard_io(A)
```

#ZERO_RAM

This directive zero's out all of the internal registers that may be used to hold variables before program execution begins.

DATA DEFINITION MENU

The following tables show the syntax for data definitions. If the keyboard TYPEDEF is used before the definition then the identifier does not allocate space but rather may be used as a type specifier in other data definitions. If the keyword CONST is used before the identifier, the identifier is treated as a constant. Constants must have an initializer and may not be changed at run-time. Pointers to constants are not permitted.

SHORT is a special type used to generate very efficient code for bit operations and I/O.
Arrays of SHORT and pointers to SHORT are not permitted.

Examples:

```
int a,b,c,d;
typedef int byte;
typedef short bit;
bit e,f;
byte g[3][2];
char *h;
enum boolean  {false, true};
boolean j;
byte k = 5;
byte const   WEEKS = 52;
byte const FACTORS [4] =
    {8, 16, 64, 128};
struct data_record  {
    byte   a [2];
    byte  b : 2; /*2 bits */
    byte  c : 3; /*3  bits*/
    int d;
}
```

Type Qualifer	
typedef	[type-qualifier] [type-specifier] [declarator]
static	Variable is globally active and initialized to 0
auto	Variable exists only while the procedure is active This is the default and AUTO need not be used.

Type-Specifier	
unsigned	Defines an 8 bit unsigned number
unsigned int	Defines an 8 bit unsigned number
int	Defines an 8 bit unsigned number
char	Defines a 8 bit character
float	Defines a 32 bit floating point number
short	Defines one bit
short int	Defines one bit
Int	By default the same as int8
long	By default the same as int16
long int	Defines a 16 bit unsigned number
signed	Defines an 8 bit signed number
signed int	Defines an 8 bit signed number
signed long	Defines a 16 bit signed number

Type-id	**An id from a TYPE definition**

Enum	An enumerated type, syntax below
Struct	A structure, syntax in next table
Union	A union, syntax in next table

declarator

```
[const]          [*]      id            [ cexpr ]      [= init]
                          ^
                          |
                          Zero or more
```

enum:

```
enum             [id]       { [ id [ = cexpr] }
                                ^
                                |
                                One or more comma separated
```

The id after ENUM is created as a type large enough to the largest constant in the list. The ids in the list are each created as a constant. By default the first id is set to zero and they increment by one. If a =cepr follows an id that id will have the value of the constant expression and the following list will increment by one.

struct and union

```
struct    [id]    {  [ type-qualifier [ [*]      id        cexpr [ cexpr ] ]]}
union                  ^                                         ^
                       |                                         |
                       One or more semi-colon                    Zero or more
```

The :cexpr after an id specifies the number of bits to use for the id. This number may be 1-8. Multiple [] may be used for multiple dimension arrays. Structures and unions may be nested. The id after STRUCT may be used in another STRUCT and the {} is not used to reuse the same structure form again.

FUNCTION DEFINITION

The format of a function definition is as follows:

qualifier id	**([** [type-specifier id **]** **)**	**{ [** stmt **] }**
^	^	^
|	|	|
Optional See Below	Zero or more comma separated. See Data Types	Zero or more Semi-colon separated. See Statements.

The qualifiers for a function are as follows:

```
VOID
type-specifier
```

A function definition may be preceded by one of the following to identify a special characteristic of the function:

```
#separate  #inline      #int_..
```

When one of the above are used and the function has a prototype (forward declaration of the function before it is defined) you must include the # directive on both the prototype and function definition.

A (non-standard) feature has been added to the compiler to help get around the problems created by the fact that pointers cannot be created to constant strings. A function that has one CHAR parameter will accept a constant string where it is called. The compiler will generate a loop that will call the function once for each character in the string.

Example:

```
void lcd_putc(char c ) {
...
}
lcd_putc ("Hi There.");
```

REFERENCE PARAMETERS MENU

The compiler has limited support for reference parameters. This increases the readability of code and the efficiency of some inline procedures. The following two procedures are the same. The one with reference parameters will be implemented with greater efficiency when it is inline.

```
funct_a(int*x,int*y){
   /*Traditional*/
   if(*x!=5)
      *y=*x+3;
}
funct_a(&a,&b);
funct_b(int&x,int&y){
   /*Reference params*/
   if(x!=5)
      y=x+3;
}
funct_b(a,b);
```

STATEMENTS

STATEMENT	EXAMPLE

if (expr) stmt; [**else** stmt;]	`if (x==25)`
	`    x=1;`
	`else`
	`    x=x+1;`

while (expr) stmt;	`while (get_rtcc()!=0)`
	`  putc('n');`

do stmt while (expr);	`do {`
	`    putc(c=getc());`
	`} while (c!=0);`

for (expr1;expr2;expr3) stmt;	`for (i=1;i<=10;++i)`
	`  printf("%u\r\n",i);`

switch (expr) {
case cexpr: stmt; //one or more **case**
[**default**:stmt]
... }
	`switch (cmd) {`
	`    case 0:  printf("cmd 0");`
	`       break;`
	`    case 1: printf("cmd 1");`
	`       break;`
	`    default: printf("bad cmd");`
	`       break; }`

return [expr];	`return (5);`
goto label;	`goto loop;`
label: stmt;	`loop: i++;`
break;	`break;`
continue;	`continue;`
expr;	`i=1;`
;	`;`

{[stmt]**}**
↑
zero or more
	`{a=1;`
	`  b=1;}`

Expressions Menu

Constants:	
123	Decimal
0123	Octal
0x123	Hex
0b010010	Binary
'x'	Character
'\010'	Octal Character
'\x	Special Character. Where x is one of: n,t,b,r,f,',\d,v,?,"
"abcdef"	String (null is added to the end)

Identifiers:	
ABCDE	Up to 32 characters beginning with a non-numeric. Valid characters are A-Z, 0-9 and _ (underscore).
ID[X]	Single Subscript
ID[X][X]	Multiple Subscripts
ID.ID	Structure or union reference
ID->ID	Structure or union reference

Expressions

In descending precedence					
(expr)					
!expr	~expr	++expr	expr++	- -expr	expr- -
(type)expr	*expr	&value	sizeof(type)		
expr*expr	expr/expr	expr%expr			
expr+expr	expr-expr				
expr<<expr	expr>>expr				
expr<expr	expr<=expr	expr>expr	expr>=expr		
expr==expr	expr!=expr				
expr&expr					
expr^expr					
expr | expr					
expr&& expr					
expr || expr					
!value ? expr: expr					
value = expr	value+=expr	value-=expr			
value*=expr	value/=expr	value%=expr			
value>>=expr	value<<=expr	value&=expr			
value^=expr	value|=expr	expr, expr			

Operators

+	Addition Operator
+=	Addition assignment operator, x+=y, is the same as x=x+y
&=	Bitwise and assignment operator, x&=y, is the same as x=x&y
&	Address operator
&	Bitwise and operator
^=	Bitwise exclusive or assignment operator, x^=y, is the same as x=x^y
^	Bitwise exclusive or operator
|=	Bitwise inclusive or assignment operator, x|=y, is the same as x=x|y
|	Bitwise inclusive or operator
?:	Conditional Expression operator
- -	Decrement
/=	Division assignment operator, x\=y, is the same as x=x/y
/	Division operator
==	Equality
>	Greater than operator
>=	Greater than or equal to operator
++	Increment
*	Indirection operator
!=	Inequality
<<=	Left shift assignment operator, x<<=y, is the same as x=x<<y
<	Less than operator
<<	Left Shift operator
<=	Less than or equal to operator
&&	Logical AND operator
!	Logical negation operator
||	Logical OR operator
%=	Modules assignment operator x%=y, is the same as x=x%y
%	Modules operator
*=	Multiplication assignment operator, x*=y, is the same as x=x*y
*	Multiplication operator
~	One's complement operator
>>=	Right shift assignment, x>>=y, is the same as x=x>>y
>>	Right shift operator
->	Structure Pointer operation
-=	Subtraction assignment operator
-	Subtraction operator

BUILT-IN FUNCTIONS COMMAND SUMMARY

Built-In Function List By Category	
RS232 I/O	**Processor Controls**
getc()	sleep()
putc()	restart_cause()
gets()	write_bank()
puts()	**I2C I/O**
printf()	i2c_start()
kbhit()	i2c_stop()
Bit/Byte Manipulation	i2C_read
shift_right()	i2c_write()
shift_left(	i2c_poll()
rotate_right()	**Timers**
rotate_left()	setup_timer_X()
bit_clear()	set_timer_X()
bit_set()	get_timer_X()
bit_test()	setup_counters()
swap()	restart_wdt()
Discrete I/O	**Standard C memory**
output_low()	memset()
output_high()	memcpy()
output_float()	**Delays**
output_bit()	delay_us()
input()	delay_ms()
set_tris_X()	delay_cycles()
Standard C Math	**Standard C Char**
abs()	atoi()
acos()	atol()
asin()	tolower()
atan()	toupper()
ceil()	isalnum()
cos()	isalpha()
exp()	isdigit()
floor()	islower()
labs(	isspace()
log()	isupper()
log10()	isxdigit()
sin()	strlen()
sqrt()	strcpy()
tan()	strncpy()
	strcmp()
	stricmp()
	strncmp()
	strcat()
	strstr()
	strchr()
	strrchr()
	strtok()
	strspn()
	strcspn()
	strpbrk()
	strlwr()

ABS
Computes the absolute value of an integer. If the result can not be represented, the behavior is undefined. This function requires STDLIB.H to be included.

ACOS
Computes the principle value of the arc cosine of the float x. Return value is in the range [0, pi] radians. For argument not in the range [-1,1] the behavior is undefined. This function requires MATH.H to be included.

ASIN
Computes the principle value of the arc sine of the float x. Return value is in the range [-pi/2 pi/2] radians. For argument not in the range [-1,1], the behavior undefined. This function requires MATH.H to be included.

ATAN
Computes the principle value of the arc tangent of the float x. Return value is in the range [-pi/2, pi/2] radians. This function requires MATH.H to be included.

ATOI
Converts the initial portion of the string pointed too by ptr to int representation. Accepts both decimal and hexadecimal argument. If the result cannot be represented, the behavior is undefined. This function requires STDLIB.H to be included.

ATOL
Converts the initial portion of the string pointed to by ptr to long int representation. Accepts both decimal and hexadecimal argument. If the result cannot be represented, the behavior is undefined. This function requires STDLIB.H to be included.

BIT_CLEAR
This function will simply clear the specified bit (0-7 or 0-15) in the given byte or word. The least significant bit is 0. This function is exactly the same as: var &= ~(1<<bit);

Example:
```
int x;
x=5;
bit_clear(x,2);// x is now 1
```

BIT_SET
This function will simply set the specified bit (0-7 or 0-15) in the given byte or word. The least significant bit is 0. This function is the same as: byte |= (1<<bit);

Example:
```
int x;
x=5;
bit_set(x,3);// x is now 13
```

BIT_TEST
This function will test the specified bit (0-7 or 0-15) in the given byte or word. The least significant bit is 0. This function is much more efficient than, but otherwise the same as: ((var & (1<<bit)) != 0)

Example:

```
if( bit_test(x,3) || !bit_test (x,1) ){
//either bit 3 is 1 or bit 1 is 0
}
```

CEIL
Computes the smallest integral value greater than the float x. This function requires MATH.H to be included.

COS
Computes the principle value of the cosine of the float x. Return is in the range [0,pi] radians. For argument not in range [-1,1] the behavior is undefined. This function requires MATH.H to be included.

DELAY_CYCLES
This function will create code to perform a delay of the specified number of instruction clocks (1-255). An instruction clock is equal to four oscillator clocks.

Example:

```
delay_cycles( 1 ); // Same as a NOP
```

DELAY_MS
This function will create code to perform a delay of the specified length. Time is specified in milliseconds and constant values may be 0-65535. Variable delays may be 0-255. Delays will be calls to a SEPARATE function. The USE DELAY must be used before this statement so the compiler knows the clock speed.

Examples:

```
#use delay (clock=20000000)
delay_ms( 2 );
void delay_seconds(int n) {
for (n!=0; n- -)
    delay_ms(  1000 );
}
```

DELAY_US
This function will create code to perform a delay of the specified length. Time is specified in microseconds and constant values may be 0-65535. Variable delays may be 0-255. Shorter delays will be INLINE code and longer delays and variable delays are calls to a SEPARATE function. The USE DELAY must be used before this statement so the compiler knows the clock speed.

Examples:

```
#use delay(clock=20000000)
delay_us(50);
delay_us(us_to_wait);
```

EXP
Computes the exponential function of the float x. If the magnitude of x is too large, the behavior is undefined. This function requires MATH.H to be included.

FLOOR
Computes the greatest integral value not greater than the float x. This function requires MATH.H to be included.

GET_RTCC

This function will return the count value of a real time clock/counter. RTCC and Timer0 are the same. Timer 1 is 16 bits and the others are 8 bits.

Example:

```
while ( get_rtcc() != 0 ) ;
```

GET_TIMER1
GET_TIMER2

This function gets the current 16 bit value of timer1 or timer2.

GET_TIMER1_CAPT(R)
GET_TIMER2_CAPT(R)

This function gets the value (16 bit) of the capture register 1 (R=1) or 2 (R=2) for either timer 1 or timer 2.

GETC
GETCH
GETCHAR

This function waits for a character to come in over the RS232 RCV pin and returns the character. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

The serial I/O procedures require the #USE DELAY in order to help determine the timing for the correct baud rate. Before the SX can be connected to a standard RS-232 device the voltage levels must be converted.

Example:

```
printf("Continue (Y,N)?");
do {
    answer=getch();
}while(answer!='Y'&&answer!='N');
```

GETS

This function reads characters (using GETC()) into the string until a RETURN (value 13) is encountered. The string is terminated with a 0. Note that INPUT.C has a more versatile GET_STRING function.

I2C_POLL

The I2C_POLL() function is only available when the built-in SSP is used. This function returns TRUE if the hardware has a received byte in the buffer. When a TRUE is returned, a call to I2C_READ() will immediately return the byte that was received.

Example:

```
i2c_start();      // Start condition
i2c_write(0xc1);  // Device address/Read
count=0;
while(count!=4) {
    if(i2c_poll())
    r[count++]= i2c_read();
                  //Read Next
                  // Do something here
}
i2c_stop();       // Stop condition
```

I2C_READ

The I2C_READ() function will read a byte over the I2C interface. A #USE I2C must be specified before the I2C()_READ call. In master mode this function will generate the clock and in slave mode it will wait for the clock. This function waits for data and no automatic timeout is provided. When RESTART_WDT is included in the #USE I2C then this function will strobe the WDT while waiting. <u>An optional parameter of 0 may be used to cause the function to not ACK the received data.</u>

Example:

```
i2c_start();      // Start condition
i2c_write(0xa1);  // Device address
r = i2c_read();   // Read first byte
r2 = i2c_read();  // Read second byte
i2c_stop();       // Stop condition
```

I2C_START

The I2C_START() function will issue a start condition when in the I2C master mode. A #USE I2C is required before this function is available. After the start condition the clock is held low until I2C_READ() and I2C_WRITE() are called. See I2C_WRITE for an example.

I2C_STOP

The I2C_STOP() function will issue a stop condition when in the I2C master mode. A #USE I2C is required before this function is available. See I2C_WRITE for an example.

I2C_WRITE

The I2C_WRITE() function will send a single byte over the I2C interface. #USE I2C must have been specified before the I2C_WRITE() call. In master mode this function will generate a clock with the data and in slave mode it will wait for the clock from the master. No automatic timeout is provided in the function when not using the built-in SSP. This function returns the ACK Bit.

Example:

```
i2c_start();     // Start condition
i2c_write(0xa0); // Device address
i2c_write(5);    // Device command
i2c_write(12);   // Device data
i2c_stop();      // Stop condition
```

INPUT

This function returns the state of the indicated pin. The method of I/O is dependent on the last USE *_IO directive. The return value is a SHORT INT.

Example:

```
while ( !input(PIN_B1) ) ;
```

Note:
The argument to the input and output functions is a bit address. For example, port a (byte 5) bit 3 would have a value of 5*8+3 or 43. This would be defined as follows:

```
#define relay_output 43
```

Pins for the devices have been already defined as PIN-xxx in the .H files provided. These may be customized to have more meaningful pin names for a given project.

ISALNUM
ISALPHA
ISDIGIT
ISLOWER
ISSPACE
ISUPPER
ISXDIGIT

CTYPE.H contains several traditional macros as follows:

Returns a TRUE if:	
ISALNUM(X)	X is 0..9, 'A'..'Z', or 'a'..'z'
ISALPHA(X)	X is 'A'..'Z' or 'a'..'z'
ISDIGIT(X)	X is '0'..'9'
ISLOWER(X)	X is 'a'..'z'
ISUPPER(X)	X is 'A'..'Z
ISSPACE(X)	X is a space
ISXDIGIT(X)	X is '0'..'9', 'A'..'F', or 'a'..'f'

KBHIT

This function will return TRUE if the start bit of a character is being sent on the RS232 RCV pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

Example:

```
keypress=' ';
while ( keypress!='Q' ) {
   if ( kbhit () )
      keypress=getc();
   if (!input(PIN_B2))
     output_high(PIN_B3);
   else
     output_low(PIN_B3);
}
```

LABS

Computes the absolute value of long integer x. If the result cannot be represented, the behavior is undefined. This function requires STDLIB.H to be included.

LOG

Computes the natural logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined. This function requires MATH.H to be included.

LOG10

Computes the base-ten logarithm of the float x. If the argument is less than or equal to zero or too large, the behavior is undefined. This function requires MATH.H to be included.

MEMCPY

This function will copy n bytes from source to dest in RAM. DEST and SOURCE must be pointers.

Example:

```
memcpy(&structA,&structB,sizeof (structA));
memcpy(arrayA,arrayB,sizeof (arrayA));
memcpy(&structA, &databyte, 1);
```

MEMSET

This function will set N bytes of memory at DEST with the VALUE. DEST must be a pointer.

Example:

```
memset(arrayA, 0, sizeof(arrayA));
memset(&structA, 0xff,sizeof(structA));
```

OUTPUT_BIT

This function will output the specified value (0 or 1) to the specified I/O pin. The method of setting the direction register is determined by the last #USE *_IO directive.

Example:

```
output_bit( PIN_B0, 0);
   // Same as output_low(pin_B0);
output_bit( PIN_B0,input( PIN_B1 ) );
   // Make pin B0 the same as B1
output_bit( PIN_B0,shift_left
   (&data,1,input(PIN_B1)));
   // Output the MSB of data to
   // B0 and at the same time
   // shift B1 into the LSB
```

OUTPUT_FLOAT

This function will set the specified pin to the input mode. This will allow the pin to float high to represent a high on an open collector type of connection.

Example:

```
if( (data & 0x80)==0 )
   output_low(pin_A0);
else
   output_float(pin_A0);
```

OUTPUT_HIGH

This procedure will set a given pin to the high state. The method of I/O used is dependent on the last USE *_IO directive.

Example:

```
output_high(PIN_A0);
```

OUTPUT_LOW

This procedure will set a given pin to the ground state. The method of I/O used is dependent on the last USE *_IO directive.

Example:

```
output_low(PIN_A0);
```

PRINTF

The formatted print (PRINTF) function will output a string of characters to either the standard RS-232 pins (first form) or to a specified function. Formatting is in accordance with the string argument. When variables are used this string must be a constant. The % character is used within the string to indicate a variable value is to be formatted and output. A %% will output a single %. The format takes the generic form %wt where w is optional and may be 1-9 to specify how many characters are to be outputted, or 01-09 to indicate leading zeros or 1.1 to 9.9 for floating point. t is the type and may be one of the following:

- C Character
- U Unsigned int

• x	Hex int (lower case output)
• X	Hex int (upper case output)
• D	Signed int
• %e	Float in exp format
• %f	Float
• Lx	Hex long int (lower case)
• LX	Hex long int (upper case)
• Iu	unsigned decimal long
• Id	signed decimal long
• %	Just a %

Examples:

```
byte  x,y,z;
printf("HiThere");
printf("RTCCValue=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",x,y,z);
printf(LCD_PUTC, "n=%v",n);
```

Example formats:

Specifer	**Value=0x12**	**Value=0xfe**
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	Fe
%X	12	FE
%4X	0012	00FE

* Result is undefined - Assume garbage.

PUTC

This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

Example:

```
if (checksum==0)
   putchar(ACK);
else
   putchar(NAK);
```

PUTCHAR

This function sends a character over the RS232 XMIT pin. A #USE RS232 must appear before this call to determine the baud rate and pin used. The #USE RS232 remains in effect until another is encountered in the file.

Example:

```
if (checksum==0)
   putchar(ACK);
else
   putchar(NAK);
```

PUTS

This function sends each character in the string out the RS232 pin using PUTC(). After the string is sent a RETURN (13) and LINE-FEED (10) are sent.

Example:

```
puts( " ----------- " );
puts( " |    HI     | " );
puts( " ----------- " );
```

READ_BANK

This function will read a data byte from the specified memory bank.   The bank may be 1- 15 and the offset may be 0-15. See the "Questions and Answers" section for more information.

Example:

```
data = read_bank(1,5);
```

RESTART_CAUSE

This function will return the reason for the last processor reset. Return values will be one of:

- WDT_FROM_SLEEP
- WDT_TIMEOUT
- MCLR_FROM_SLEEP
- NORMAL_POWER_UP

Example:

```
switch ( restart_cause() ) {
    case WDT_FROM_SLEEP:
    case WDT_TIMEOUT:
            handle_error();
}
```

READ_COMP

This function reads the comparator status and sets the mode to one of:

• COMP_DISABLE	Comparator off
• COMP_ENABLE	Comparator on
• COMP_ENABLE|COMP_RB0	On and result output on RB0

Example:

```
READ_COMP(comp_enable)
            .
            .
            .
if(read_comp(comp_enable) & 1)
            Printf ("B2 > B1");
```

READ_CONTROL

This function reads a processor control register. Group must be 5-9 and mode may be 0-1F.

Example:

```
TICNTA=read_control(6,7);
```

RESTART_WDT

This function will restart the watchdog timer. If the watchdog timer is enabled, this must be called periodically to prevent the processor from resetting.

Example:

```
while (!done) {
restart_wdt();
   .
   .
   .
}
```

ROTATE_LEFT

This function will rotate a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bytes is the number of bytes involved in the rotate operation. Bit 0 of the lowest BYTE in RAM is considered the LSB.

Example:

```
x = 0x86;
rotate_left( &x, 1);
// x is now 0x0d
```

ROTATE_RIGHT

This function will rotate a bit through an array or structure. The address may be an array identifier or an address to a byte or structure (such as &data). Bytes is the number of bytes involved in the rotate operation. Bit 0 of the lowest BYTE in RAM is considered the LSB.

Example:

```
struct {
int cell_1 : 4;
int cell_2 : 4;
int cell_3 : 4;
int cell_4 : 4; } cells;
rotate_right( &cells, 2);
rotate_right( &cells, 2);
rotate_right( &cells, 2);
rotate_right( &cells, 2);
// cell_1->4, 2->1, 3->2 and 4-> 3
```

SET_MODE_A
SET_MODE_B
SET_MODE_C
SET_MODE_D
SET_MODE_E

This function sets the mode of each I/O pin on a given port. Each pin maybe:

• mOUTPUT	Output Pin
• mTTLIN	Input with TTL Levels
• mSTIN	Smitt trigger input
• mCMOSIN	Input with CMOS levels
• mPULLUP	Or with an input to enable pullups

Example:

```
SET_MODE_B (mOUTPUT,mOUTPUT, mTTLIN,
mTTLIN,MTTLIN,mTTLIN|mPULLUP,mSTIN,mOUTPUT);
```

SETUP_MIWU

This function sets the wakeup/interrupt mode on the port B pins. Each pin maybe one of:

• miwuL2H	Interrupt on low to high
• miwuH2L	Interrupt on high to low
• miwuOFF	No action

Example:

```
SETUP_MIWU (miwuOFF, miwuOFF, miwuOFF, miwuOFF, miwuOFF, miwuL2H,
miwuL2H, miwuL2H);
```

SET_RTCC

This function will set a timer to the specified value. RTCC and Timer0 are the same. Timer1 is 16 bits and the others are 8 bits.

Example:

```
                if ( get_rtcc()==25 )set_rtcc(0);
```

SET_TIMER1
SET_TIMER2

This function sets the timer (1 or 2) to the 16 bit value.

SET_TRIS_A
SET_TRIS_B
SET_TRIS_C
SET_TRIS_D
SET_TRIS_E

These functions allow the tri-state registers to be written to directly. This must be used with FAST_IO and when I/O ports are accessed as memory such as when a #BYTE directive is used.

Each bit in the value represents one pin. A 1 indicates the pin is input and a 0 indicates it is output.

Example:

```
                SET_TRIS_B( 0x0F );
```

SETUP_COUNTERS

This function sets up the RTCC or WDT. The rtcc_state determines what drives the RTCC. The PS state sets a pre-scaler for either the RTCC or WDT. The pre-scaler will lengthen the cycle of the indicated counter. If the RTCC pre-scaler is set the WDT will be set to WDT_18MS. If the WDT pre-scaler is set the RTCC is set to RTCC_DIV_1.

rtcc_state values:

- RTCC_INTERNAL
- RTCC_EXT_L_TO_H
- RTCC_EXT_H_TO_L

ps_state values:

- RTCC_DIV_2
- RTCC_DIV_4
- RTCC_DIV_8
- RTCC_DIV_16
- RTCC_DIV_32
- RTCC_DIV_64
- RTCC_DIV_128
- RTCC_DIV_256
- WDT_18MS
- WDT_36MS
- WDT_72MS
- WDT_144MS
- WDT_288MS
- WDT_576MS
- WDT_1152MS

- WDT_2304MS

Example:

```
setup_counters (RTCC_INTERNAL, WDT_2304MS);
```

SETUP_TIMER1
SETUP_TIMER2

This function sets up the specified timer. Mode may be in:

tSW	Software
tPWM	Pulse Width Modulator
tCC	Capture/Compare
tEXT	External Counter

And may be OR'ed with:

CAPT_INT	Interrupt on capture
COMP_INT	Interrupt on compare
OVFL_INT	Interrupt on overflow
CAPT_L2H	Capture on low to high
CAPT_H2L	Capture on high to low
EXT_L2H	Ext input on low to high
EXT_H2L	Ext input on high to low

Prescale may be:

DIV_BY_1	DIV_BY_16
DIV_BY_2	DIV_BY_32
DIV_BY_4	DIV_BY_64
DIV_BY_8	DIV_BY_128

COMP1 and COMP2 are the values for the two compare registers.

SHIFT_LEFT

This function will shift a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &data). Bytes is the number of bytes involved in the shift operation. Value is the bit value to insert. This function will return the bit that is shifted out. Bit 0 of the lowest BYTE in ROM is treated as the LSB.

Example:

```
byte buffer[3];
for(I=1; I<=24; ++I){
    while (!input(PIN_A2));
    shift_left(buffer,3,input(PIN_A3));
    while (input(PIN_A2));
}
/* reads 24 bits from pin A3,each bit is read on a low to high on
pin A2 */
```

SHIFT_RIGHT

This function will shift a bit into an array or structure. The address may be an array identifier or an address to a structure (such as &data). Bytes is the number of bytes involved in the shift operation. Value is the bit value to insert. This function will return the bit that is shifted out. Bit 0 of the lowest BYTE in RAM is treated as the LSB.

Example:

```
struct {
byte time;
    byte command : 4;
    byte source  : 4;} msg;
for(i=0; i<=16; ++i) {
    while(!input(PIN_A2));
    shift_right(&msg,3,input(PIN_A1));
    while (input(PIN_A2)) ;}
/* reads 16 bits from pin A1, each bit is read on a low to high on
pin A2 */
for(i=0;i<8;++i)
    output_bit(PIN_A0,shift_right
    (&data,1,0));
/*This shifts 8 bits out PIN_A0, LSB first.*/
```

SIN
Computes the principle value of the sine of the float x. Return value is in the range [-pi/2, pi/2] radians. This function requires MATH.H to be included.

SLEEP
This function will issue a SLEEP instruction.

Example:
```
SLEEP();
```

SQRT
Computes the non-negative square root of the float x. If the argument is negative, the behavior is undefined.

STANDARD STRING FUNCTIONS
These functions are defined in the string.h include file which must be included before they may be used. Use STRCPY to copy a constant string to a RAM based string before using the following functions.

CHAR * STRCAT (char *s1, char *s2)
Appends copy of string s2 to end of s1, and returns a pointer to new s1

CHAR * STRCHR (char *s, char c)
Finds first occurrance of character c in s and returns a pointer to it

CHAR * STRRCHR (char *s, char c)
Finds last occurrence of character c in s and returns a pointer to it.

SIGNED INT STRCMP (char *s1, char *s2)
Compares s1 and s2 and returns -1 if s1<s2, 0 if s1=s2, and 1 if s1>s2.

SIGNED INT STRNCMP (char *s1, char *s2, int n)
Compares max of n characters (not following '0') from s1 to s2; returns the same as STRCMP.

SIGNED INT STRICMP (char *s1, char *s2)
Compares s1 to s2 while ignoring the case. (UPPER vs. lower); returns the same as STRCMP.

CHAR * STRNCPY (char *s1, char *s2, int n)
Copies max of n characters (not following ending '0') from s2 in s1; if s2 has less than n characters, it appends '0' at the end.

INT STRCSPN (char *s1, char * s2)
Computes length of max initial segment of s1 that consists entirely of characters NOT from s2.

INT STRSPN (char *s1, char *s2)
Computers length of max initial segment of s1 consisting entirely of characters from s2.

INT STRLEN (char *s)

Computes length of s (excluding terminating '\0').

CHAR * STRLWR (char *s)
Replaces uppercase letters with lowercase and returns a pointer to s.

CHAR * STRPBRK (char *s1, char *s2)
Locates first occurrence of any character from s2 in s1 and returns a pointer to it; returns s1 if s2 is an empty string.

CHAR * STRSTR (char *s1, char *s2)
Locates first occurrence of character sequence s2 in s1 and returns pointer to it; returns null if s2 is an empty string.

CHAR * STRTOK (char *s1, char *s2)
Finds next token in s1 delimited by a character from separator string s2 (which can be different from call to call), and returns pointer to it.

First call starts at beginning of s1 searching for the first character NOT contained in s2 and returns null if there is none is found.

If none are found, it is the start of first token (return value). Function then searches from there for a character contained in s2.

If none are found, current token extends to the end of s1, and subsequent searches for a token will return null.

If one is found, it is overwritten by '\0', which terminates current token. Function saves pointer to following character from which next search will start.

Each subsequent call, with 0 as first argument, starts searching from the saved pointer.

STRCPY
Copies a constant string to RAM.

Example:
```
char string[10];
.
.
.
strcpy (string, "Hi There");
```

SWAP
This function will swap the upper nibble with the lower nibble of the specified byte. This is the same as:
byte = (byte << 4) | (byte >> 4);

Example:
```
x=0x45;
swap(x);//x now is 0x54
```

TAN
Computes the principle value of the tangent of the float x. Return value is in the range [-pi/2, pi/2] radians. This function requires MATH.H to be included.

TOLOWER
TOUPPER

TOLOWER(X) will return 'a'..'z' for every 'A'..'Z' and all other characters are unchanged.
TOUPPER(X) will return 'A'..'Z' for every 'a'..'z' and all other characters are unchanged.

WRITE_BANK

This function will write a data byte to the specified memory bank. See the "Questions and Answers" section for more information.

Example:

```
WRITE_BANK(1,0,23);
```

WRITE_CONTROL

This function writes to a processor control register. Group may be 5-9 and mode may be 0-1F.

Example:

```
write_control (6,15,0);
```

Same as:

```
set_tris_B(0); on the SX20AC
```

COMPILER ERROR MESSAGES

#ENDIF with no corresponding #IF
A numeric expression must appear here. The indicated item must evaluate to a number.

Array dimensions must be specified
The [ ] notation is not permitted in the compiler. Specific dimensions must be used. For example A[5].

Arrays of bits are not permitted
Arrays may not be of SHORT INT. Arrays of Records are permitted but the record size is always rounded up to the next byte boundary.

Attempt to create a pointer to a constant
Constant tables are implemented as functions. Pointers can not be created to functions. For example CHAR CONST MSG[9]={"HI THERE"}; is permitted, however you can not use &MSG. You can only reference MSG with subscripts such as MSG[i] and in some function calls such as Printf and STRCPY.

Attributes used may only be applied to a function (INLINE or SEPARATE)
An attempt was made to apply #INLINE or #SEPARATE to something other than a function.

Bad expression - Must be constant
The indicated expression must evaluate to a constant at compile time. For example 5*3+1 is permitted but 5*x+1 where X is a INT is not permitted. If X were a DEFINE that had a constant value then it is permitted.

Bad expression syntax
This is a generic error message. It covers all incorrect syntax.

Baud rate out of range
The compiler could not create code for the specified baud rate. If the internal UART is being used the combination of the clock and the UART capabilities could not get a baud rate within 3% of the requested value. If the built in UART is not being used then the clock will not permit the indicated baud rate. For fast baud rates, a faster clock will be required.

BIT variable not permitted here
Addresses can not be created to bits. For example &X is not permitted if X is a SHORT INT.

Can't change device type this far into the code
The #DEVICE is not permitted after code is generated that is device specific. Move the #DEVICE to an area before code is generated.

Character constant constructed incorrectly
Generally this is due to too many characters within the single quotes. For example 'ab' is an error as is '\nr'. The backslash is permitted provided the result is a single character such as '\010' or '\n'.

Constant out of the valid range
This will usually occur in inline assembly where a constant must be within a particular range and it is not. For example BTFSC 3,9 would cause this error since the second operand must be from 0-8.

Constant too large, must be < 65536
As it says the constant is too big.

Define expansion is too large
A fully expanded DEFINE must be less than 255 characters. Check to be sure the DEFINE is not recursively defined.

Define syntax error
This is usually caused by a missing or mis-placed (or) within a define.

Different levels of indirection
This is caused by a INLINE function with a reference parameter being called with a parameter that is not a variable. Usually calling with a constant causes this.

Divide by zero
An attempt was made to divide by zero at compile time using constants.

Duplicate DEFAULT statements
The DEFAULT statement within a SWITCH may only appear once in each SWITCH. This error indicates a second DEFAULT was encountered.

Duplicate function
A function has already been defined with this name. Remember that the compiler is not case sensitive unless a #CASE is used.

Duplicate Interrupt Procedure
Only one function may be attached to each interrupt level. For example the #INT_RB may only appear once in each program.

Duplicate USE
Some USE libraries may only be invoked once since they apply to the entire program such as #USE DELAY. These may not be changed throughout the program.

Element is not a member
A field of a record identified by the compiler is not actually in the record. Check the identifier spelling.

ELSE with no corresponding IF
Check that the {and} match up correctly.

End of file while within define definition
The end of the source file was encountered while still expanding a define. Check for a missing).

End of source file reached without closing comment */ symbol
The end of the source file has been reached and a comment (started with /*) is still in effect. The */ is missing.

Error in define syntax

Error text not in file
The error is a new error not in the error file on your disk. Check to be sure that the errors.txt file you are using came on the same disk as the version of software you are executing. Call CCS with the error number if this does not solve the problem.

Expect ;
Expect comma
Expect WHILE
Expect }
Expecting :

Expecting =
Expecting a (
Expecting a , or)
Expecting a , or }
Expecting a .
Expecting a ; or ,
Expecting a ; or {
Expecting a close paren
Expecting a declaration
Expecting a structure/union
Expecting a variable
Expecting a]
Expecting a {
Expecting an =
Expecting an array
Expecting an expression
Expecting an identifier

Expecting an opcode mnemonic
This must be a Scenix mnemonic such as MOVLW or BTFSC.

Expecting LVALUE such as a variable name or * expression
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.

Expecting a basic type
Examples of a basic type are INT and CHAR.

Expecting procedure name

Expression must be a constant or simple variable

Expression must evaluate to a constant

Expression too complex
This expression has generated too much code for the compiler to handle for a single expression. This is very rare but if it happens, break the expression up into smaller parts.

Extra characters on preprocessor command line
Characters are appearing after a preprocessor directive that do not apply to that directive. Preprocessor commands own the entire line unlike the normal C syntax. For example the following is an error:

```
#PRAGMA DEVICE <SX28AC> main() { int x; x=1;}
```

File in #INCLUDE can not be opened
Check the filename and the current path. The file could not be opened.

Filename must start with " or <

Filename must terminate with " or >

Floating-point numbers not supported
A floating-point number is not permitted in the operation near the error. For example, ++F where F is a float is not allowed.

Function definition different from previous definition

This is a mis-match between a function prototype and a function definition. Be sure that if a #INLINE or #SEPARATE are used that they appear for both the prototype and definition. These directives are treated much like a type specifier.

Function used but not defined
The indicated function had a prototype but was never defined in the program.

Identifier is already used in this scope
An attempt was made to define a new identifier that has already been defined.

Illegal C character in input file
A bad character is in the source file. Try deleting the line and re-typing it.

Improper use of a function identifier
Function identifiers may only be used to call a function. An attempt was made to otherwise reference a function. A function identifier should have a (after it.

Incorrectly constructed label
This may be an improperly terminated expression followed by a label. For example:
x=5+

Initialization of unions is not permitted
Structures can be initialized with an initial value but UNIONS can not be.

Internal compiler limit reached
The program is using too much of something. An internal compiler limit was reached. Contact CCS and the limit may be able to be expanded.

Invalid conversion from LONG INT to INT
In this case, a LONG INT can not be converted to an INT. You can type cast the LONG INT to perform a truncation. For example:
I = INT(LI);

Internal Error - Contact CCS
This error indicates the compiler detected an internal inconsistency. This is not an error with the source code; although, something in the source code has triggered the internal error. This problem can usually be quickly corrected by sending the source files to CCS so the problem can be re-created and corrected.
In the meantime if the error was on a particular line, look for another way to perform the same operation. The error was probably caused by the syntax of the identified statement. If the error was the last line of the code, the problem was in linking. Look at the call tree for something out of the ordinary.

Invalid parameters to shift function
Built-in shift and rotate functions (such as SHIFT_LEFT) require an expression that evaluates to a constant to specify the number of bytes.

Invalid Pre-Processor directive
The compiler does not know the preprocessor directive. This is the identifier in one of the following two places:
#xxxxx
#PRAGMA xxxxx

Library in USE not found
The identifier after the USE is not one of the pre-defined libraries for the compiler. Check the spelling.

LVALUE required
This error will occur when a constant is used where a variable should be. For example 4=5; will give this error.

Macro identifier requires parameters
A #DEFINE identifier is being used but no parameters were specified ,as required. For example:
#define min(x,y) ((x<y)?x:y)
When called MIN must have a (—-,—-) after it such as:
r=min(value, 6);

Missing #ENDIF
A #IF was found without a corresponding #ENDIF.

Must have a #USE DELAY before a #USE RS232
The RS232 library uses the DELAY library. You must have a #USE DELAY before you can do a #USE RS232.

No MAIN() function found
All programs are required to have one function with the name main().

Not enough RAM for all variables
The program requires more RAM than is available. The memory map (ALT-M) will show variables allocated. The ALT-T will show the RAM used by each function. Additional RAM usage can be obtained by breaking larger functions into smaller ones and splitting the RAM between them.

For example, a function A may perform a series of operations and have 20 local variables declared. Upon analysis, it may be determined that there are two main parts to the calculations and many variables are not shared between the parts. A function B may be defined with 7 local variables and a function C may be defined with 7 local variables. Function A now calls B and C and combines the results and now may only need 6 variables. The savings are accomplished because B and C are not executing at the same time and the same real memory locations will be used for their 6 variables (just not at the same time). The compiler will allocate only 13 locations for the group of functions A, B, C where 20 were required before to perform the same operation.

Number of bits is out of range
For a count of bits, such as in a structure definition, this must be 1-8. For a bit number specification, such as in the #BIT, the number must be 0-7.

Out of ROM, A segment or the program is too large
A function and all of the INLINE functions it calls must fit into one segment (a hardware code page). If a program has only one function and that function is 600 instructions long, you will get this error even though the chip has plenty of ROM left. The function needs to be split into at least two smaller functions. Even after this is done, this error may occur since the new function may be only called once and the linker might automatically INLINE it. This is easily determined by reviewing the call tree via ALT-T. If this error is caused by too many functions being automatically INLINED by the linker, simply add a #SEPARATE before a function to force the function to be SEPARATE. Separate functions can be allocated on any page that has room. The best way to understand the cause of this error is to review the calling tree via ALT-T.

Pointers to bits are not permitted
Addresses can not be created to bits. For example, &X is not permitted if X is a SHORT INT.

Pointers to functions are not valid
Addresses can not be created to functions.

Previous identifier must be a pointer
A -> may only be used after a pointer to a structure. It can not be used on a structure itself or other kind of variable.

Printf format type is invalid
An unknown character is after the % in a printf. Check the printf reference for valid formats.

Printf format (%) invalid
A bad format combination was used. For example, %lc.

Printf variable count (%) does not match actual count
The number of % format indicators in the printf does not match the actual number of variables that follow. Remember in order to print a single %, you must use %%.

Procedure definition different from previous definition

Recursion not permitted
The linker will not allow recursive function calls. A function may not call itself and it may not call any other function that will eventually re-call it.

Recursively defined structures not permitted
A structure may not contain an instance of itself.

Reference arrays are not permitted
A reference parameter may not refer to an array.

String too long

Structures and UNIONS can not be parameters (use * or &)
A structure may not be passed by value. Pass a pointer to the structure using &.

Subscript out of range
A subscript to a RAM array must be at least 1 and not more than 128 elements. Note that large arrays might not fit in a bank. ROM arrays may not occupy more than 256 locations.

This expression can not evaluate to a number
A numeric result is required here and the expression used will not evaluate to a number.

This type can not be qualified with this qualifier
Check the qualifiers. Be sure to look on previous lines. An example of this error is:
VOID X;

Too many #DEFINE statements
The internal compiler limit for the permitted number of defines has been reached. Call CCS to find out if this can be increased.

Too many array subscripts
Arrays are limited to 5 dimensions.

Too many constant structures to fit into available space
Available space depends on the chip. Some chips only allow constant structures in certain places. Look at the last calling tree to evaluate space usage. Constant structures will appear as functions with a @CONST at the beginning of the name.

Too many identifiers have been defined

The internal compiler limit for the permitted number of variables has been reached. Call CCS to find out if this can be increased.

Too many identifiers in program
The internal compiler limit for the permitted number of identifiers has been reached. Call CCS to find out if this can be increased.

Too many nested #INCLUDEs
No more than 10 include files may be open at a time.

Too many parameters
More parameters have been given to a function than the function was defined with.

Too many subscripts
More subscripts have been given to an array than the array was defined with.

Type is not defined
The specified type is used but not defined in the program. Check the spelling.

Type specification not valid for a function
This function has a type specifier that is not meaningful to a function.

Undefined identifier
The specified identifier is being used but has never been defined. Check the spelling.

Undefined label that was used in a GOTO
There was a GOTO LABEL but LABEL was never encountered within the required scope. A GOTO can not jump outside a function.

Unknown device type
A #DEVICE contained an unknown device. The center letters of a device are always C regardless of the actual part in use. Be sure the correct compiler is being used for the indicated device. See #DEVICE for more information.

Unknown keyword in #FUSES
Check the keyword spelling against the description under #FUSES.

Unknown type
The specified type is used but not defined in the program. Check the spelling.

USE parameter invalid
One of the parameters to a USE library is not valid for the current environment.

USE parameter value is out of range
One of the values for a parameter to the USE library is not valid for the current environment.

COMMON QUESTIONS AND ANSWERS

How does one input or output an entire byte to an I/O port?

The INPUT, OUTPUT_HIGH and OUTPUT_LOW functions operate on a single pin. To operate on an entire port, one of the following schemes can be used:

```
#byte   PORTB  =  6
#define ALL_OUT 0
#define ALL_IN  0xff
main() {
   int i;

   set_tris_b(ALL_OUT);
   PORTB = 0;// Set all pins low
   for(i=0;i<=127;++i)  // Quickly count from 0 to 127
          PORTB=i;      // on the I/O port pin
   set_tris_b(ALL_IN);
   i = PORTB;           // i now contains the portb value.
}
```

Remember when using the #BYTE, the created variable is treated like memory. You must maintain the tri-state control registers yourself via the SET_TRIS_X function. Following is an example of placing a structure on an I/O port:

```
struct    port_b_layout
                 {int data : 4;
                 int rw : 1;
                 int cd : 1;
                 int enable : 1;
                 int reset  : 1; };
struct    port_b_layout  port_b;
#byte            port_b  = 6
struct    port_b_layout  const  INIT_1  = {0, 1,1,1,1};
struct    port_b_layout  const  INIT_2  = {3, 1,1,1,0};
struct    port_b_layout  const  INIT_3  = {0, 0,0,0,0};
struct    port_b_layout  const  FOR_SEND = {0,0,0,0,0};
                                       // All outputs
struct    port_b_layout  const  FOR_READ = {15,0,0,0,0};
                                       // Data is an input
main() {
   int x;
   set_tris_b((int)FOR_SEND);   // The constant
                                // structure is
                                // treated like
                                // a byte and
                                // is used to
                                // set the data
                                // direction
   port_b = INIT_1;
   delay_us(25);
port_b = INIT_2;                // These constant structures
delay_us(25);                   // are used to set all fields
   port_b = INIT_3;             // on the port with a single
                                // command
   set_tris_b((int)FOR_READ);
   port_b.rw=0;
                                // Here the individual
   port_b.cd=1;                 // fields are accessed
   port_b.enable=0;             // independently.
   x = port_b.data;
   port_b.enable=0
}
```

Why does a program work with standard I/O but not with fast I/O?

First remember that the fast I/O mode does nothing except the I/O. The programmer must set the tri-state registers to establish the direction via SET_TRIS_X(). The SET_TRIS_X() function will set the direction for the entire port (8 bits). A bit set to 1 indicates input and 0 is an output. For example, to set all pins of port B to outputs except the B7 pin, use the following:

```
set_tris_b( 0x80 );
```

Secondly, be aware that fast I/O can be very fast. Consider the following code:

```
output_high( PIN_B0 );
output_low( PIN_B1 );
```

This will be implemented with two assembly instructions (SETB 6,0 and CIRB 6,1). The microprocessor implements the BSF and BCF as a read of the entire port, a modify of the bit and a write back of the port. In this example, at the time that the BCF is executed, the B0 pin may not have yet stabilized. The previous state of pin B0 will be seen and written to the port with the B1 change. In effect, it will appear as if the high to B0 never happened. With standard and fixed I/O, this is not usually a problem since enough extra instructions are inserted to avoid a problem. The time it takes for a pin to stabilize depends on the load placed on the pin. The following is an example of a fix to the above problem:

```
output_high( PIN_B0 );
delay_cycles(1);   //Delay one instruction time
output_high( PIN_B1 );
```

The delay_cycles(1) will simply insert one NOP between the two I/O commands. At 20mhz a NOP is 0.2 us.

Why does the generated code that uses BIT variables look so ugly?

Bit variables (SHORT INT) are great for both saving RAM and for speed but only when used correctly. Consider the following:

```
int x,y;
short int bx, by;
x=5;
y=10;
bx=0;
by=1;
x = (x+by)-bx*by+(y-by);
```

When used with arithmetic operators (+ and - above), the BX and BY will be first converted to a byte internally: this is ugly. If this must be done, you can save space and time by first converting the bit to byte only once and saving the compiler from doing it again and again. For example:

```
z=by;
x = (x+z)-bx*z+(y-z);
```

Better, would be to avoid using bits in these kinds of expressions. Almost always, they can be rewritten more efficiently using IF statements to test the bit variables. You can make assignments to bits, use them in IFs and use the &&, || and ! operators very efficiently. The following will be implemented with great efficiency:

```
If (by || (bx && bz) || !bw)
z=0;
```

Remember to use ! not ~, && not & and || not | with bits. Note that the INPUT(...) function and some other built-in functions that return a bit follow the same rules. For example do the following:

```
if ( !input( PIN_B0 ) )
```

NOT:

```
if( input( PIN_B0 ) == 0)
```

Both will work but the first one is implemented with one bit test instruction and the second one does a conversion to a byte and a comparison to zero.

Why is the RS-232 not working right?

1. The SX is Sending Garbage Characters.
A. Check the clock on the target for accuracy. Crystals are usually not a problem but RC oscillators can cause trouble with RS-232. Make sure the #USE DELAY matches the actual clock frequency.

B. Make sure the PC (or other host) has the correct baud and parity setting.

C. Check the level conversion. When using a driver/receiver chip, such as the MAX 232, do not use INVERT when making direct connections with resistors and/or diodes. You probably need the INVERT option in the #USE RS232.

D. Remember that PUTC(6) will send an ASCII 6 to the PC and this may not be a visible character. PUTC('A') will output a visible character A.

2. The SX is Receiving Garbage Characters.
A. Check all of the above.

3. Nothing is Being Sent.
A. Make sure that the tri-state registers are correct. The mode (standard, fast, fixed) used will be whatever the mode is when the #USE RS232 is encountered. Staying with the default STANDARD mode is safest.

B. Use the following main() for testing:

```
main() {
    while(TRUE)
            putc('u');
}
```

Check the XMIT pin for activity with a logic probe, scope or whatever you can. If you can look at it with a scope, check the bit time (it should be 1/BAUD). Check again after the level converter.

4. Nothing is being received.
First be sure the SX can send data. Use the following main() for testing:

```
main() {
    printf("start");
      while(TRUE)
         putc( getc()+1 );
}
```

When connected to a PC typing A should show B echoed back.
If nothing is seen coming back (except the initial "Start"), check the RCV pin on the SX with a logic probe. You should see a HIGH state and when a key is pressed at the PC, a pulse to low. Trace back to find out where it is lost.

5. The SX is always receiving data via RS-232 even when none is being sent.
A. Check that the INVERT option in the USE RS232 is right for your level converter. If the RCV pin is HIGH when no data is being sent, you should NOT use INVERT. If the pin is low when no data is being sent, you need to use INVERT.

B. Check that the pin is stable at HIGH or LOW in accordance with A above when no data is being sent.

C. When using PORT A with a device that supports the SETUP_PORT_A function make sure the port is set to digital inputs. This is not the default. The same is true for devices with a comparator on PORT A.

6. Compiler reports INVALID BAUD RATE.
A. When using a software RS232 (no built-in UART), the clock cannot be really slow when fast baud rates are used and cannot be really fast with slow baud rates. Experiment with the clock/baud rate values to find your limits.

How can I use two or more RS-232 ports on one chip?

The #USE RS232 (and I2C for that matter) is in effect for GETC, PUTC, PRINTF and KBHIT functions encountered until another #USE RS232 is found.

The #USE RS232 is not an executable line. It works much like a #DEFINE.
The following is an example program to read from one RS-232 port (A) and echo the data to both the first RS-232 port (A) and a second RS-232 port (B).

```
#USE  RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
void  put_to_a( char c ) {
   put(c);
}
char  get_from_a( ) {
          return(getc()); }
#USE RS232(BAUD=9600, XMIT=PIN_B2,RCV=PIN_B3)
void put_to_b( char b ) {
   putc(c);
}
main() {
   char c;
   put_to_a("Online\n\r");
   put_to_b("Online\n\r");
   while(TRUE) {
     c=get_from_a();
     put_to_b(c);
     put_to_a(c);
   }
}
```

The following will do the same thing but is less readable:

```
main()  {
char c;
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    printf("Online\n\r");
#USE RS232(BAUD=9600, #useXMIT=PIN_B2,RCV=PIN_B3)
    printf("Online\n\r");
    while(TRUE) {
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    c=getc();
#USE RS232(BAUD=9600, XMIT=PIN_B2,RCV=PIN_B3)
    putc(c);
#USE RS232(BAUD=9600, XMIT=PIN_B0, RCV=PIN_B1)
    putc(c);
}
```

What can be done about an OUT OF RAM error?

The compiler makes every effort to optimize usage of RAM. Understanding the RAM allocation can be a help in designing the program structure. The best re-use of RAM is accomplished when local variables are used with lots of functions. RAM is re-used between functions not active at the same time. See the NOT ENOUGH RAM error message in this manual for a more detailed example.

RAM is also used for expression evaluation when the expression is complex. The more complex the expression, the more scratch RAM locations the compiler will need to allocate to that expression. The RAM allocated is reserved during the execution of the entire function but may be re-used between expressions within the function. The total RAM required for a function is the sum of the parameters, the local variables and the largest number of scratch locations required for any expression within the function. The RAM required for a function is shown in the call tree after the RAM=. The RAM stays used when the function calls another function and new RAM is allocated for the new function. However when a function RETURNS the RAM may be re-used by another function called by the parent. Sequential calls to functions each with their own local variables is very efficient use of RAM as opposed to a large function with local variables declared for the entire process at once.

Be sure to use SHORT INT (1 bit) variables whenever possible for flags and other boolean variables. The compiler can pack eight such variables into one byte location. This is done automatically by the compiler whenever you use SHORT INT. The code size and ROM size will be smaller.

Finally, consider an external memory device to hold data not required frequently. An external 8 pin EEPROM or SRAM can be connected to the SX with just 2 wires and provide a great deal of additional storage capability. The compiler package includes example drivers for these devices. The primary drawback is a slower access time to read and write the data. The SRAM will have fast read and write with memory being lost when power fails. The EEPROM will have a very long write cycle, but can retain the data when power is lost.

Why does the .LST file look out of order?

The list file is produced to show the assembly code created for the C source code. Each C source line has the corresponding assembly lines under it to show the compiler's work. The following three special cases make the .LST file look strange to the first time viewer. Understanding how the compiler is working in these special cases will make the .LST file appear quite normal and very useful.

1. Stray code near the top of the program is sometimes under what looks like a non-executable source line.
Some of the code generated by the compiler does not correspond to any particular source line. The compiler will put this code either near the top of the program or sometimes under a #USE that caused subroutines to be generated.

2. The addresses are out of order.
The compiler will create the .LST file in the order of the C source code. The linker has re-arranged the code to properly fit the functions into the best code pages and the best half of a code page. The resulting code is not in source order. Whenever the compiler has a discontinuity in the .LST file, it will put a * line in the file. This is most often seen between functions and in places where INLINE functions are called. In the case of a INLINE function, the addresses will continue in order up where the source for the INLINE function is located.

3. The compiler has gone insane and generated the same instruction over and over.

For example:

```
...........A=0;
03F:        CLR     15
*
46: CLR     15
*
051:        CLR     15
*
113:        CLR     15
```

This effect is seen when the function is an INLINE function and is called from more than one place. In the above case, the A=0 line is in a INLINE function called in four places. Each place it is called from gets a new copy of the code. Each instance of the code is shown along with the original source line, and the result may look unusual until the addresses and the * are noticed.

How does the compiler handle converting between bytes and words?

In an assignment such as:

```
bytevar = wordvar;
```

The most significant BYTE is lost. This is the same result as:

```
bytevar = wordvar & 0xff;
```

The following will yield just the most significant BYTE:

```
bytevar = wordvar >> 8;
```

Any arithmetic or relational expression involving both bytes and words will perform word operations, and treat the bytes as words with the top byte 0. For example:

```
wordvar= 0x1234;
bytevar= 0x34;
if(wordvar==bytevar)     //will be FALSE
```

Any arithmetic operations that only involve bytes will yield a byte result even when assigned to word. For example:

```
bytevar1 = 0x80;
bytevar2 = 0x04;
wordvar = bytevar1 * bytevar2;
//wordvar will be 0
```

However, typecasting may be used to force word arithmetic:

```
wordvar = (long) bytevar1 * (long) bytevar2;

//wordvar will be 0x200
```

How does the compiler determine TRUE and FALSE on expressions?

When relational expressions are assigned to variables, the result is always 0 or 1.
For example:

```
bytevar = 5>0;     //bytevar will be 1
bytevar = 0>5;     //bytevar will be 0
```

The same is true when relation operators are used in expressions. For example:

```
bytevar = (x>y)*4;
```

is the same as:

```
if( x>y )
  bytevar=4;
else
  bytevar=0;
```

SHORT INTs (bit variables) are treated the same as relational expressions. They evaluate to 0 or 1.When expressions are converted to relational expressions or SHORT INTs, the result will be FALSE (or 0) when the expression is 0, otherwise the result is TRUE (or 1).
For example:

```
bytevar = 54;
bitvar = bytevar; //bitvar will be 1 (bytevar ! = O)
if(bytevar)              //will be TRUE
bytevar = 0;
bitvar = bytevar; //bitvar will be 0
```

What are the restrictions on function calls from an interrupt function?

Whenever interrupts are used, the programmer MUST ensure there will be enough stack space. The compiler checks the size of the stack used by the main() function and will not allow you to exceed it. Interrupts, however, use additional stack space. One location for the interrupt function and one location for each separate call from the interrupt function are used. The call tree may be reviewed in the IDE. Ensure the size of the stack required by the interrupt plus the size of the stack already used by main() wherever interrupts are enabled is less than 9.

The compiler does not permit recursive calls to functions because the RISC instruction set does not provide an efficient means to implement a traditional C stack. All RAM locations required for a given function are allocated to a specific address at link time in such a way that RAM is re-used between functions not active at the same time. This prohibits recursion. For example, the main() function may call a function A() and A() may call B() but B() may NOT call main(), A() or B().

An interrupt may come in at any time, which poses a special problem. Consider the interrupt function called ISR() that calls the function A() just like main() calls A(). If the function A() is executing because main() called it and then the ISR() activates, recursion will have happened.
In order to prevent the above problem, the compiler will "protect" the function call to A() from main() by disabling all interrupts before the call to A() and restoring the interrupt state after A() returns. In doing so, the compiler can allow complete sharing of functions between the main program and the interrupt functions.

The programmer must take the following special considerations into account:
1. In the above example, interrupts will be disabled for the entire execution of A(). This will increase the interrupt latency depending on the execution time of A().

2. If the function A() changes the interrupts using ENABLE/DISABLE _INTERRUPTS then the effect may be lost upon the return from A(), since the entire INTCON register is saved before A() is called and restored afterwards. Furthermore, if the global interrupt flag is enabled in A(), the program may execute incorrectly.

3. A program should not depend on the interrupts being disabled in the above situation. The compiler may NOT disable interrupts when the function or any function it calls requires no local RAM.

4. The interrupts may be disabled, as described above for internal compiler functions called by the same manor. For example, multiplication invoked by a simple * may have this effect.

Instead of 400, the compiler calls 0. Why?

The SX ROM address field in opcodes is 9 Bits. The rest of the address bits come from other sources. For example, call address 400 from code in the first page you will see:

```
Page 400
CALL      0
```

The call 0 is actually 400H.

Instead of 3F, the compiler is using register 1F. Why?

The RAM address field in opcodes is 5 bits long, depending on the chip. The rest of the address field comes from the status register. For example, to load 3F you will see:

```
Bank       20
MOV 1F,W
```

Note that the bank may not be immediately before the access since the compiler optimizes out the redundant bank switches.

How do I directly read/write to internal registers?

A hardware register may be mapped to a C variable to allow direct read and write capability to the register. The following is an example using the TIMER0 register:

```
#BYTE timer0 = 0x01
timer0= 128; //set timer0 to 128
while (timer0 ! = 200); // wait for timer0 to reach 200
```

Bits in registers may also be mapped as follows:

```
#BIT CARRY = 3.0
.
.
.
while (!CARRY);
```

Registers may be indirectly addressed as shown in the following example:

```
printf ("enter address:");
a = gethex ();
printf  ("\r\n value is %x\r\n", *a);
```

The compiler has a large set of built-in functions that will allow one to perform the most common tasks with C function calls. When possible, it is best to use the built-in functions rather than directly write to registers. Register locations change between chips and some register operations require a specific algorithm to be performed when a register value is changed. The compiler also takes into account known chip errata in the implementation of the built-in functions. For example, it is better to do set_tris_A(0); rather than *0x85=0;

How can a constant data table be placed in ROM?

The compiler has support for placing any data structure into the device ROM as a constant read-only element. Since the ROM and RAM data paths are separate in the SX, there are restrictions on how the data is accessed. For example, to place a 10 element BYTE array in ROM use:

```
BYTE CONST TABLE [10]= {9,8,7,6,5,4,3,2,1,0};
```

and to access the table use:

```
x = TABLE [i];
```

OR

```
x = TABLE [5];
```

BUT NOT

```
ptr = &TABLE [i];
```

In this case, a pointer to the table cannot be constructed.

Similar constructs using CONST may be used with any data type including structures, longs and floats.

Note that in the implementation of the above table, a function call is made when a table is accessed with a subscript that cannot be evaluated at compile time.

How can the RB interrupt be used to detect a button press?

The RB interrupt will happen when there is any change (input or output) on pins B4-B7. There is only one interrupt and the SX does not tell you which pin changed. The programmer must determine the change based on the previously known value of the port. Furthermore, a single button press may cause several interrupts due to bounce in the switch. A debounce algorithm will need to be used. The following is a simple example:

```
#int_rb
rb_isr ( ) {
    byte changes;
    changes = last_b ^ port_b;
    last_b = port_b;
    if (bit_test(changes,4 )&& !bit_test(last_b,4)){
         //b4 went low
}
if (bit_test(changes,5)&& !bit_test (last_b,5)){
          //b5 went low
    }
    .
    .
    .
    delay-ms (100);  //debounce
}
```

The delay=ms (100) is a quick and dirty debounce. In general, you will not want to sit in an ISR for 100 MS to allow the switch to debounce. A more elegant solution is to set a timer on the first interrupt and wait until the timer overflows. Don't process further changes on the pin.

Why does the compiler show less RAM than there really is?

The compiler has two methods of handling pointers (#DEVICE *=5) and (#DEVICE *=8). The default is *=8. The modes are as follows:

***=5**

Pointers may only be 0-1F. The compiler allocates no variables over 1F but you can use #BYTE or Read/Write_Bank to access the memory. This is the most efficient code but reduces easily usable RAM.

***=8**

Pointers may be 0-255. The compiler allocates variables up to 255. Additional code is generated on every pointer access due to the chip design. Memory in bank 0, locations 0-F on BD ports are only accessed via #BYTE at addresses 100-10F.

Memory not available for automatic allocations is not counted in totals and percentages.

EXAMPLE PROGRAMS

A large number of example programs are included on the disk. The following is a list of many of the programs and some of the key programs are re-printed on the following pages. Most programs will work with any chip by just changing the #INCLUDE line that includes the device information. All of the following programs have wiring instructions at the beginning of the code in a comment header. The SIOW.EXE program included in the program directory may be used to demonstrate the example programs. This program will use a PC COM port to communicate with the target.
Generic header files are included for the standard SX parts. These files are in the DEVICES directory. The pins of the chip are defined in these files in the form PIN_B2. It is recommended that for a given project, the file is copied to a project header file and the PIN_xx defines be changed to match the actual hardware. For example; LCDRW (matching the mnemonic on the schematic). Use the generic include files by placing the following in your main .C file:
#include <SX28AC.H>

SX_SQW.C
This is a short program that uses RS-232 I/O to talk to a user and upon command will be in a 1khz square wave. This simple program shows how easy it is to use the basic built-in functions.

SX_PULSE.C
This program will use the RTCC (timer0) to time a single pulse input to the SX. This program will show how to use the RTCC and how to output a decimal number.

SX_ADMM.C
This simple A/D program takes 30 A/D samples and displays the minimum and maximum values over the RS-232. The process is forever repeated.

SX_STWT.C
This program uses interrupts to keep a real time seconds counter. It then implements a stopwatch function over the RS-232. This program will show how simple it is to use interrupts.

SX_LCDKB.C
This program uses both a 16x2 LCD module and a 3x4 keypad to demonstrate the use of the **LCD.C** driver and **KBD.C** driver. This program will display keypad input on the LCD.

SX_EXTEE.C
This is a general-purpose serial EEPROM read/write program. This may be used with a large number of devices depending on the include file used. The following include files have fully tested drivers for various devices:
2401.C, 2402.C, 2404.C, 2408.C, 2416.C, 2432.C, 2465.C, 9346.C, 9356.C, 9366.C, 9356SPI.C
The 24xx.C files demonstrate the use of I2C. The other files demonstrate doing serial I/O over 2/3 wire interfaces. The files with SPI at the end use the internal SSP hardware.

SX_RTC.C
This program uses the RS-232 interface to read and set an external RTC chip. One of the following chips may be used by simply including the correct INCLUDE file in the program:
DS1302.C, NJU6355.C

SX_RTCLK.C
Same as SX_RTC.C except the interface to the RTC is using a LCD and keypad.

SX_AD12.C
Similar to SX.ADMM but uses a 12 bit external A/D part. This program uses the **LTC1298.C** driver.

SX_STEP.C
This is an example program to drive a stepper motor.

SX_X10.C
This program demonstrates the **X10.C** driver by providing a X10 to RS-232 interface. X10 codes will be sent to the PC and may be entered at the PC for transmission.

SX_TOUCH.C
This program uses the **TOUCH.C** driver to show how easy it is to interface to the Dallas touch devices.

SX_SRAM.C
This program may be used with either the **DS2223.C** or **PCF8570.C** drivers to interface to an external serial RAM chip. This is a great way to gain additional memory.

SX_DEC.C
This example shows how to create your own special purpose formatted print functions for types such as fixed point.

SX_FLOAT.C
Demonstrates the compiler's floating point capability.

SX_PWM.C
This program shows how to use the built-in PWM with the compiler built-in functions.

SX_CCP1S.C
This program uses the hardware CCP to generate a precision one-shot pulse.

SX_CCPMP.C
This program uses the H/W CCP and compiler built-ins to measure a pulse width.

SX_LED.C
This program directly drives a two-digit 7-segment LED display.

SX_TEMP.C
This program uses the **DS1621.C** driver to display the temperature in Fahrenheit.

SX_PBUSM.C
This program shows how to set up a one-wire SX to a SX message program.

SX_PBUSR.C
This program shows how to set up a one-wire SX to SX shadow RAM. Any number of SXs may be connected on the wire and each will have a RAM block. When the RAM is changed in one SX, it will be changed in all other SXS.

SX_SISR.C
This program shows how to implement an interrupt-driven RS232 receiver.

SX_EXPIO.C
This program uses **74165.C** and **74595.C** to show how to use external chips to create more input and output pins.

SX_DPOT.C
This program uses **DS1868.C** to show how to implement a digital POT.

SX_LNG32.C
This program shows how to do 32 bit math using math.c.

SX_1920.C
An example of a Dallas 1920 temp sensor.