A short guide to the Embed Inc PIC development environment

The contents of this document were created from the different text files found on the Embed Inc WEB site. Embed Inc is in no way responsible for any errors that might have been added in the conversion from the text files to SDML.

This document was formatted using DECdocument on OpenVMS. Conversion from Postscript to PDF was done using Ghostscript on OpenVMS.

The HTML to SDML conversion and generation of this PDF file was done by Jan-Erik Söderholm, S:t Anna Data, Söderköping, Sweden. Any comments about this PDF file can be sent to jan-erik.soderholm@telia.com.

Last change: 22-Sep-2003.

Contents

GENERAL INFORMATION

EXAMPLES

TABLES

General information

This part of the document contains information collected from the EmbedInc web site. As little as possible is changed, to make future revisions due to changes to the environment easier.

1 EmbedInc PIC Development

1.1 Structure of PIC Firmware

Embed Inc is a custom embedded systems development company. We have done many projects using the Microchip PIC line of microcontrollers and have converged on a structure and methodology for PIC projects that is suitable for the majority of cases. This includes a set of standard macros, subroutines, modules, and host tools.

When possible, we use our common framework for PIC firmware projects. The framework has evolved over many projects, and will continue to evolve. It consists of two master include files, STD_DEF.INS.ASPIC and STD.INS.ASPIC, a common naming scheme and structure for the modules of a project, and a few standard modules for capabilities required by most PIC projects.

Unfortunately, the Microchip tools only allow DOS file names. This means that names are limited to 8 characters, a dot, and 3 more characters. The last three characters are used to indicate the file type, so you are only left with 8 characters to distinguish one source file from another. We use the convention PPP_MMMM.ASPIC. PPP stands for the 3 letter project ID, and MMMM is the maximum of 4 letter module name within the project. For example, "hal_intr.aspic" is the source file for the INTR module within the HAL project. Yes, this sucks. Hopefully Microchip will update their tools soon.

So what's with this ASPIC stuff? We use a preprocessor on all our PIC source code. The .aspic file name suffix indicates to our build system that it is a PIC assembler module that must be run thru the preprocessor to create a .asm file. The .asm file produced by the preprocessor is assembled directly by the MPASM PIC assembler.

Below are the standard include files, and templates for the standard modules and additional include files. The project name of the templates is "qqq". The strings "qqq" and "QQQ" in the template files are intended to be replaced with the actual project name via global substitution in a text editor.

Table 1–1 Standard modules and files

File	Use
STD_DEF.INS.ASPIC	This is the first of the two "mandatory" include files used by all projects. This file sets defaults that will be used in STD.INS.ASPIC. The two files are separated so that the application can change the defaults before they are used.

Table 1–1 (Cont.) Standard modules and files

File	Use
STD.INS.ASPIC	This is the second of the two "mandatory" include files. This is the file that does all the work, which is mostly defining a set of macros. It relies on the configuration values originally set in STD_DEF.INS.ASPIC, which can be altered by the application before this file is included.
QQQLIB.INS.ASPIC	This is the top include file for the project and declares the PIC environment. This include file references STD_DEF.INS.ASPIC, customizes specific values, then includes STD.INS.ASPIC. Non-environment issues related to the operation of the project are in qqq.INS.ASPIC. Standard library routines used by this project only use this include file and not qqq.INS.ASPIC. Modules specific to the project include qqq.INS.ASPIC, which includes qqqLIB.INS.ASPIC.
QQQ.INS.ASPIC	Project-specific include file. All project definitions except those relating to just the PIC environment go here.
QQQ_STRT.ASPIC	Top executable module. This module contains the reset vector and initializes each of the modules individually. When done, it jumps to the INIT entry point in the INIT module.
QQQ_INIT.ASPIC	INIT in this module is jumped to after all the modules have been intialized individually. This module performs system-level initialization and jumps to MAIN in the MAIN module.
QQQ_MAIN.ASPIC	Main operating code. Entry point MAIN is jumped to after initialization is complete.
QQQ_PORT.ASPIC	Low level I/O port handling. The PORT_INIT routine initializes the I/O ports as defined by the /INBIT and /OUTBIT preprocessor directives in qqq.INS.ASPIC. This module also disables any analog inputs the chip might have, which are usually enabled on power up. If used, these would be specifically enabled by the modules that handle them. Initializing all analog inputs to off eliminates some common bugs in porting from a PIC without an analog peripheral to one that has an analog peripheral.
QQQ_INTR.ASPIC QQQ_INTR18.ASPIC	Interrupt handlers. QQQ_INTR18.ASPIC for the 18 family. QQQ_INTR18.ASPIC is configurable to single or multiple priority interrupts by setting the INTR_PRIORITIES constant at the top of the file to true or false.
QQQ_UART.ASPIC	Fully interrupt driven UART handler with software receive and transmit FIFOs.
QQQ_CMD.ASPIC	Host command interpreter. The command interpreter runs as a separate thread. It is run for a time slice from the main event loop when a UART input byte is availble (FLAG_SIN set).
QQQ_XXXX.ASPIC	Template for an arbitrary project module.

In general, each module contains the qqq_INIT routine, which performs any module-specific initialization. The qqq_INIT routines are called from the STRT module after a reset.

Table 1–2 Additional modules and files

File	Use
HAL example project	This project follows the structure and naming conventions described above. It is the PIC code for the SpookyEyes halloween device. See the comments in HAL.INS.ASPIC for more details.
HOS example project	Template project that acts on commands sent from a host or dumb terminal via RS-232.

1.2 PIC Assembler Build Tools

We use wrapper programs around each of the Microchip build tools: MPASM, MPLIB, and MPLINK. This is partly because the Microchip tools are too brain dead to return a bad program exit status when they fail. Our build system, like most build systems, requires a tool to return something other than 0 program exit status when the build failed. Our wrappers also deal with some file naming and other issues.

In addition to plain wrappers for the Microchip executables, we have created a Microchip assembler preprocessor. This allows us to add directives to the assembler source code that enable better programming practises than raw MPASM does. See the doc file for details.

Here is the list of some of the PIC development tools.

Table 1–3 PIC Development tools

Tool	Use
PREPIC	PIC assembler preprocessor. Required by ASMPIC if assembling .ASPIC files.
ASMPIC	Wrapper around MPASMWIN. Requires PREPIC if assembling .ASPIC files.
LIBPIC	Wrapper around MPLIB.
LINKPIC	Wrapper around MPLINK.

2 General information about Embed Inc software downloads

This page provides general information about Embed Inc software downloads.

2.1 Self-extracting executables

Most software releases on this web site are provided as self-extracting executable files. The files are named INSTALL_xxx.EXE, where XXX is a name for the specific software release. When run, these files will extract themselves into a temporary directory to create an Embed Inc software release directory, then automatically run the installation program from that directory. You will be prompted where the software is to be installed before any changes are made to your system. You can abort the software installation process at the first prompt by closing the window with the prompts. In that case, the temporary software release directory will be deleted and no net change will have been made to your system.

To install the software, follow the directions as prompted by the installation program. The temporary software release directory will be deleted when the installation procedure completes.

After clicking on a INSTALL_xxx.EXE file name, you can save the program to disk or run it directly. Saving it to disk is useful if you want to install the software on other systems without having to download it from the web each time. Running it directly is the easiest way to download and install the software on the machine you are on, but the installation program will not be saved for re-use later or on other machines.

2.2 Installing multiple releases

Multiple Embed Inc software releases can be installed together, and must be installed to the same directory if the combined features are to be available. The installation program will notify you if a previous Embed Inc software installation is found, and set the default installation directory so that the new release is merged into the existing installed Embed Inc software. The default software installation directory will be c:\embedinc if no previous installation was found.

If you install multiple software releases, install them in order according their release dates as shown on the page you downloaded them from. Install old releases first.

If installing multiple software releases together, *be sure to reboot if directed after installing the first one.* A software release is not fully installed until the machine is rebooted if this was requested by the installation program. Subsequent releases may not install correctly until the first one is fully installed, including the required reboot.

Additional releases can then be installed without rebooting between each one, although the machine must be rebooted to complete the combined installation if a reboot was requested by *any* of the additional installations.

2.3 Structure of installed software

All Embed Inc software has a common structure after installation. This section describes the different top level directories within the software installation directory.

Note: Only some of these directories may exist in any one software release. For example, if a software release contains no source code, then it will not contain a SOURCE directory.

The default software installation directory is c:\embedinc. The directories described here are found directly within this installation directory. For example, the full path to the COM directory will be c:\embedinc\com if the software was installed to the default location.

The top level directories within a software installation directory are:

Table 2–1 Install directories

Directory	Use
COM	Executables (commands). These include shell scripts (.bat files). This directory is added to the command search path by the installation procedure. Most of the programs are intended to be run from a command line.
DOC	Documentation files. In general, documentation files are plain text files, and there is one documentation file here for each program in the COM directory. There are also additional documentation files that are not specific to particular programs. The DOC command, if present, will look for a documentation file in this directory or a .BAT file in the COM directory and display it. It can be useful to look thru the DOC directory and .BAT files in the COM directory after installing a software release.
ENV	Environment files. These are used implicitly by many programs and you generally shouldn't mess with them.
EXAMPLES	This directory contains sub-directories with the same names as programs in the COM directory. Each such sub-directory contains example files that are related to that program. Many programs don't have example files, and will therefore not have a sub-directory here. You can safely modify anything in the EXAMPLES directory without effecting the behaviour of any programs.
FONTS	Vector font files used by graphics programs. Stay out of here.
LIB	Linkable libraries and their associated include files.
PROGS	Like the EXAMPLES directory, this contains sub-directories with the same names as programs in the COM directory. This is where files required for a program's operation go. You should think of these files as part of the program itself and leave them alone.
SOURCE	Source code, which is defined as those files used to produce software that are directly modified by a human, usually with a text editor. This directory contains only subdirectories, which contain the actual source code files.

Table 2–1 (Cont.) Install directories

Directory	Use
SRC	There is one directory here for each directory in SOURCE. These directories contain the files derived from the raw source code in SOURCE. All builds are run in these directories. The SOURCE directory contains the pure source code, and everything else gets created here. If everything is set up correctly, you should be able to delete all files in a SRC sub-directory and recreate them solely from the source code in the SOURCE directories.

2.4 Operating systems

The following operating systems are supported:

- Windows NT 4
- Windows 2000
- Windows XP

2.4.1 Special note for Win9x users

The Win9x operating systems (Windows 95, 98, and ME) are obsolete relics of a bygone era, and we have no intention of supporting them. The software itself will actually run on these systems, but the automated installation procedure will not. If you think you know better and run one of the installation executables on Win9x anyway, don't come complaining to us about the resulting mess. (Yes that sounds stupid, but it has actually happened.)

If you absolutely insist on clinging to Win9x, you can install a software release manually. Here is what to do:

1 Run the INSTALL_xxx.EXE installation executable for the software you want to install. This will pop up a command shell window with a message something like:

```
Software installation source directory is C:\temp\WZSE0.TMP\embedinc.

No previous installation of this software was detected.

Enter where you want the software installed, or hit ENTER to choose the
default shown in parenthesis.
(C:\embedinc):
```

DO NOT PRESS ENTER or type anything else in the window. The purpose of this step is only to extract the data files from the self-extracting executable.

2 Look at the first sentence in the command shell window to see where the data files got extracted to. In the above example that is C:\temp\WZSE0.TMP\embedinc. The directory one level up is a full Embed Inc software installation directory, which is C:\temp\WZSE0.TMP in this example.

3 The installation directory will contain a file called readme.txt (C:\temp\WZSE0.TMP\readme.txt in this example). Read the directions for the Win9x operating systems.

4 If you chose to continue, follow the directions except do not yet reboot the machine.

5 Go to the command shell window that popped up from step 1 above. Enter CTRL-C at the prompt only. *DO NOT PRESS ENTER*. Entering CTRL-C will kill the window and delete the temporary installation directory from your disk.

6 Reboot the machine as directed in the last step of the instructions in the readme.txt file.

3 Embed Inc software downloads

This chapter describes the various software releases for download.

Be sure to read the legal notice before using any software found on the Embed Inc. web site.

3.1 PIC development tools

install_picdev.exe (1,018,368 bytes, 8 April 2003)

This is a stand-alone release containing all the PIC software development tools. This release includes:

- PREPIC
- ASMPIC
- LIBPIC
- LINKPIC
- Various linker control files in SOURCE/PIC.
- A few related utility programs.
- Source code for the QQQ_xxx module template files in SOURCE/PIC.
- Source code for the mandatory PIC include files in SOURCE/PIC.
- The minimum required PIC library files, REGS.INC and STACK.INC in SRC/PIC.

Note: Note that you probably need to set the MPLABDIR environment variable to run any of the MPLAB tools from the development environment. See the ASMPIC documentation file for details.

3.2 HAL example PIC project

install_pic_hal.exe (292,352 bytes, 12 June 2002)

This is an incremental release that adds the source code and build script for the HAL PIC project to the installed software.

Note: This software release will not work on its own. At a minimum, it requires the PIC development tools (above) to also be installed.

The HAL source code is in the PICS source directory, and is built with the BUILD_HAL_EXPIC script in the COM directory.

3.3 HOS template PIC project

install_pic_hos.exe (145,920 bytes, 10 December 2002)

This is an incremental release that adds the source code and build script for the HOS PIC project to the installed software.

Note: This software release will not work on its own. At a minimum, it requires the PIC development tools (above) to also be installed.

The HOS source code is in the PICS source directory, and is built with the BUILD_HOS_EXPIC script in the COM directory.

3.4 WAV file manipulation

install_wav.exe (985,600 bytes, 12 June 2002)

This is a stand-alone release containing programs for manipulating WAV files and related utilities. This release includes, in part:

- WAV_COPY - Copies a WAV file and can apply modifications in the process.
- WAV_INFO - Shows info about a WAV file.
- WAV_CSV - Creates a CSV (comma separated values) file from a WAV file.
- CSVPLOT - Plots the contents of a CSV (comma separated values) file.

3.5 Chess program

install_chess_exe.exe (478,208 bytes, 12 January 2003)

This is a stand alone release of a computer chess program. This release only contains the minimum runtime files of the visual (as apposed to text only) chess program CHESSV.

This software was designed to let others install private position evaluators without having to re-invent the GUI, legal move generator, etc. We plan on making the customizable version available here when the necessary documentation and release/install scripts have been written.

For now just run CHESSV and have fun. There is no documentation, but you should be able to figure out everything by browsing the menus.

4 Syntax of the xxxxPIC programs

The following pages describes the syntax of ASMPIC, PREPIC, LIBPIC and LINKPIC

ASMPIC

This program is a wrapper around the Microchip MPASMWIN assembler. The assembler always sets the program status code to zero, whether errors were encountered or not. It does, however, produce a .ERR file which is empty if all went well and contains error messages when all did not go well. This file is used to set the exit status of ASMPIC.

SYNTAX

ASMPIC *source file [opt1] ... [optN]*

PARAMETERS

source file

the name of the source file to assemble. The source file name must end in ".asm" if it is in native Microchip PIC assembler format, or ".aspic" if it is in preprocessor format. In the latter case, the preprocessor is run to produce the .asm file, which is then passed to the Microchip assembler. The input file name suffix may be omitted, in which case a .aspic file is used in preference to a .asm file if both are present.

opt1 to optN

Options according to the option list.

DESCRIPTION

The /q switch is always passed to the assembler, and should not be passed on the command line. This causes the assembler to run in "batch" mode without requiring user input.

The /e switch is always passed to the assembler, and should not be passed on the command line. This causes the assembler to create an error output file if errors are encountered. This program will not function correctly if the error output file is disabled. The error output file is the only evidence the assembler provides as to whether assembly was successful or not. The return status from ASMPIC is derived from the error output file, which is always deleted before ASMPIC returns.

The assembler will be run in the directory containing the source file. This is also where output files will be put by default. If errors are found, they will be listed to standard output and the program exit status will be set to ERROR. If no errors are encountered, the program exit status will be OK. The .ERR file is deleted in any case.

Note: The environment variable MPLABDIR is assumed to be set to the directory containing the Microchip executables. If this variable is not present or empty, then the Microchip executables are assumed to be in the executables search path, in other words they can be run directly without having to specify the full path name.

SWITCHES

/a hex_format

Set the HEX output file format. HEX output files are only produced by the assembler in absolute mode. In relocatable mode the linker produces the HEX file. Choices of HEX file format are INHX8M, INHX8S, and INHX32. The default is INHX8M.

/c+
/c-

Enable (+) or disable (-) case sensitivity. Case sensitivity is enabled by default.

Note: Note that most of the symbols in the Microchip processor-specific include files are upper case. This means they have to be entered in the source code in upper case when case sensitivity is enabled.

/d[symbol] [=value]

Define a symbolic constant. This has the same effect as an EQU directive defining the same constant at the beginning of the file.

/l+
/l-
/lfilename

Enable, disable, or set listing output file. The default is to produce a listing file with the same generic pathname as the input file.

/m+
/m-

Enable or disable macro expansion in the listing file. Macro expansion is enabled by default.

/o+
/o-
/ofilename

Enable, disable, or set object output file. Enabling object file output also sets relocatable, as apposed to absolute, mode. Object file output is disabled by default (absolute mode).

/p[processor]

Set processor type. The processor types are the PIC model names, like "PIC16C54" for example.

/r[radix]

Set the default radix for numeric constants. Choices are HEX, DEC, or OCT. The default is HEX.

/t[size]

Set listing file tab column widths.

/w[level]

Set output messages level. Choices of LEVEL are:

Table 4–1 Level valus to the /w switch

Value	Meaning
0	Show all messages. Default value
1	Show only error and warning messages.
2	Show only error messages.

/x+
/x-
/xfilename

Enable, disable, or set cross reference file. Cross reference file output is disabled by default.

PREPIC

Preprocess a Microchip PIC assembler source file to produce a file that can be passed directly to the assembler.

SYNTAX

PREPIC *input file name [output file name]*

PARAMETERS

input file name

The input file name must be in the format [name].aspic or [name].ins.aspic.

output file name

The output file name will end with .asm in the first case and with .inc in the second case. The file name suffixes may be omitted on the command line. By default, the output file is written to the current directory.

The input file is just copied to the output file except when a preprocessor command or inline function is encountered.

PREPROC COMMANDS

General

Lines containing preprocessor commands must have the first non-blank character be a slash (/), followed directly by a preprocessor command name, which may be followed by command parameters. The command name and parameters are separated from each other by one or more spaces. Preprocessor command names are not case sensitive.

Unless otherwise noted, each preprocessor command line is written to the output file as a comment line immediately preceeding its expansion, if any. The preprocessor commands are:

/INBIT name port bit [PUP]

Declares a particular I/O pin as an input.

Table 4–2 INBIT parameters

Parameter	Description
NAME	Name being defined for the I/O pin, will be used to generate unique assembler symbols associated with this pin, shown below.
PORT	Address of the port register containing the I/O bit. For example, "PORTA" or "PORTC". Some processors have a single I/O port call GPIO. On these processors, a PORTA alias has been created so that that the PORT argument can always have the form "PORTx".
BIT	Bit number within the register.

Table 4–2 (Cont.) INBIT parameters

Parameter	Description
PUP	An optional keyword indicating that the internal passive pullup should be initially enabled for this I/O pin. An error will be generated when the standard PORT module is assembled if the requested combination of passive pullups requested by all /INBIT commands is not attainable on this specific processor.

This preprocessor command only creates or modifies assembler state. It generates no executable code. It is assumed that the assembler state has been properly initialized prior to this /INBIT command, as is done if the standard include files STD_DEF.INS.ASPIC and STD.INS.ASPIC are used as directed. The actual runtime initialization code that uses the assembler state is in the PORT_INIT subroutine in the standard PORT module.

The following assembler symbols will be defined or modified:

Table 4–3 INBIT assembler symbols

Symbol	Description
<name>_reg	Assembler constant equal to the address of the PORTx register containing the direct I/O pin value.
<name>_tris	Assembler constant equal to the address of TRISx register controlling the I/O pin in/out direction.
<name>_lat	Assembler constant equal to the address of LATx register containing the output latch bit for this I/O pin. This constant is only defined if the machine has a port latch register. For example, 16 family PICs do not have these, although 18 family PICs do.
<name>_bit	Assembler constant equal to the bit number for this I/O pin within the PORTx, TRISx, and other possible port-related registers.
VAL_TRISx	Assembler variable containing the initial value for the TRISx register (where X is the port name, like "A") controlling the direction of this I/O bit. The bit for this I/O pin will be set, which will cause the pin to be initialized as an input by the standard PORT_INIT subroutine.
VAL_PULLUPx	Assembler variable indicating the desired initial state of the passive pullups for the port containing the I/O bit. The "x" in the variable name is the port name, like "B" or "C" for example. A bit in this variable is set to indicate the associated passive pullup is to be initially enabled. Illegal requests for passive pullups will generate an error message when the standard PORT module is assembled.

The following assembler macros will be defined :

Table 4–4 INBIT macros

Const	Description
<name>_pin	String substitution macro that expands to the port register followed by the bit number within that register for this I/O pin. The string substitution macro is defined: `#define <name>_pin <name>_reg,<name>_bit` This string substitution macro can be used to supply both arguments to bit manipulation instructions as shown in the example below.
<name>_pinlat	String substitution macro that expands to the latch register followed by the bit number within that register for this I/O pin. This is like the <name>_pin macro (above), except that it refers to the bit in the latch instead of the port register. This macro is only defined when the latch register exists.

Example 4–1 Using the /INBIT preprocessor command

The following preprocessor command:

```
/inbit button portb 3 pup  ;low when user button is pressed
```

declares bit 3 of port B (RB3 pin) to be an input with its passive pullup enabled. The equivalent of the following assembler statements will be generated:

```
button_reg  equ  portb
button_tris equ  trisb
button_lat  equ  latb          (only on machines that have LATB)
button_bit  equ  3
#define button_pin portb,3
#define button_pinlat latb,3   (only on machines that have LATB)
val_trisb   set  val_trisb | b'00001000'
val_pullupb set  val_pullupb <bar> b'00001000'
```

Example code using these declarations is:

```
dbankif button_reg  ;set banks for access to BUTTON pin
btfsc   button_pin  ;is the button pressed ?
goto    done_button ;no, done handling user button
;
;   The user button is pressed.
;
...
done_button unbank       ;done handling the user button
```

Note: Undefined I/O pins default to outputs driven low.

/OUTBIT name port bit [val]

Just like /INBIT except that it declares the I/O bit to be an output instead of an input. This means the VAL_TRISp assembler variable will be updated to indicate this bit is an output.

The optional VAL parameter is the value this bit should be set to when the I/O ports are initialized. VAL must be either 0 or 1. The VAL_PORTp assembler variable will be updated with its appropriate bit set to VAL. The default for VAL is 0.

Note: Undefined I/O pins default to outputs driven low.

/FLAG name

Create a new global flag with the indicated name. The assembler variable NFLAGB is updated if a new byte is required to hold the new flag. The flags will be in bytes named GFLn, where N starts at 0 and increases by one as each new flag byte is required. The following assembler constants will be defined:

Table 4–5 FLAG assmembler constants

Const	Description
flag_[name]_regn	Number of the GFLn register containing the flag bit. In other words, this constant will be 0 if the flag is in GFL0, 1 if it is in GFL1, etc.
flag_[name]_bit	Number of the flag bit within the GFL0 register. The least significant bit is numbered 0, and the most significant is 7.

A string substitution macro is created called flag_[name] which expands into the byte containing the flag and the bit number within the byte. The string substitution macro is written:

```
#define flag_[name] gfl[n],[bit]
```

Example 4–2 Using the /FLAG preprocessor command

```
/flag    t100    ;set every 100mS
.
.
.
dbankif gbankadr
btfss   flag_t100   ;time to do 100mS processing ?
goto    done_100ms  ;no, skip this section
bcf     flag_t100   ;reset to 100mS proc not pending
;perform 100mS processing here
done_100ms unbank   ;done with the 100mS processing
```

INLINE FUNCTIONS

General

The preprocessor replaces each inline function with the expansion of that function. These inline functions can appear anywhere except within quoted strings surrounded by quotes ("") or apostrophies (’), or within comments. The format of an inline function is:

```
[name arg arg ... arg]
```

Note: Note that the brackets ([ ]) are literal.

NAME is the name of the function, and is always case-insensitive. Functions can have zero or more arguments, depending on the function. The function name and the arguments are separated by one or more spaces. Any number of spaces are allowed immediately after the "[" and immediately before the "]". String arguments must be enclosed in quotes ("") or apostrophies (’). The source characters starting with the "[" and ending with the "]" are replaced by the function value in the output file.

Inline functions can be nested to any depth. The functions are evaluated and replaced with their expansions from the innermost to the outermost.

Each function argument has one of four data types. These are:

Table 4–6 PREPIC inline function argument types

Arg type	Description
Boolean	This is always either TRUE or FALSE, although these keywords are case-insensitive.
Integer	A consecutive string of decimal digits.
Floating point	A consecutive string of decimal digits containing exactly one decimal point before, within, or after it. This string may be optionally followed by "E" and an exponent value. This is all interpreted as you would expect if you’ve used a computer since the late 1950s.
String	A string of characters enclosed in quotes ("") or apostrophies (’). The value is the characters between but not including the quotes.

The inline functions are:

FP24I val

Converts the numeric value VAL to PIC 24 bit floating point and returns the result as a 6 character hex integer in native assembler format. For example:

```
[FP24I 3.14159] --> h’419220’
```

STR arg ... arg

Returns the concatenation of all the arguments after they are converted to strings. Floating point arguments are converted to strings containing 6 significant digits. If no arguments are given, then the result is the empty string. For example:

```
[str "abc" 13 'def' 27.1] --> abc13def27.1000
```

QSTR characters

Returns a quoted string.

Note: This function does not follow the normal rules for arguments. The characters starting 2 columns after QSTR up to the "]" is enterpreted as one text string whether it contains blanks or not.

For example:

```
[qstr This is a simple test.] --> "This is a simple test."
```

FFTC2 tcfilt sfreq

Computes the filter fraction of a single pole low pass filter. TCFILT is the 1/2 decay time of the filter, and SFREQ is the sample frequency. The filter fraction is the weighting fraction of the incoming signal each filter iteration. The result is always a floating point constant.

FFFREQ ffreq sfreq

Computes the filter fraction of a single pole low pass filter. FFREQ is the desired -3dB rolloff frequency of the filter, and SFREQ is the sample frequency.

SIN ang

Returns the floating point sine of the angle ANG in radians.

COS ang

Returns the floating point cosine of the angle ANG in radians.

TAN ang

Returns the floating point tangent of the angle ANG in radians.

PI

Returns the value of Pi in floating point, 3.141592653589...

RDEG ang

Converts the angle ANG from radians to degrees. The result data type is floating point.

DEGR ang

Converts the angle ANG from degrees to radians. The result data type is floating point.

+ arg ... arg

Computes the sum of all the arguments. The result is integer 0 if no arguments are supplied. The result data type is integer if all arguments are integers, and floating point if one or more arguments are floating point.

- arg1 arg2

Computes the subtraction of ARG1 minus ARG2. The result data type is integer if all arguments are integers, and floating point if one or more arguments are floating point.

** arg ... arg*

Computes the product of all the arguments. The result is integer 1 if no arguments are supplied. The result data type is integer if all arguments are integers, and floating point if one or more arguments are floating point.

/ arg1 arg2

Computes ARG1 divided by ARG2. The arguments may be either integer or floating point. The result data type is floating point.

DIV arg1 arg2

Performs an integer divide of ARG1/ARG2. Both arguments must be integer. The result data type is integer.

RND arg

Returns ARG rounded to the nearest integer.

TRUNC arg

Returns ARG rounded to the nearest integer towards zero. This has the effect of "truncating" the fraction part of the value.

LIBPIC

This program runs the Microchip library manager MPLIB. MPLIB requires all arguments on the command line, including the list of object files to put into a library.

This program runs MPLIB in the directory where the input file is stored. Each line in the input file is added as one additional argument to MPLIB. The input file is deleted if MPLIB completes without errors.

Note: The environment variable MPLABDIR is assumed to be set to the directory containing the Microchip executables. If this variable is not present or empty, then the Microchip executables are assumed to be in the executables search path, in other words they can be run directly without having to specify the full path name.

SYNTAX

LIBPIC *filename*

PARAMETERS

filename
The first file name is the name of the library to manipulate. Additional file names form the list of members.

INPUT FILE COMMANDS

/c
Create a new library file.

/t
Lists the members of a library.

/d
Delete members from a library.

/r
Replace members in a library. If a member does not previously exist in a library then it is added.

/x
Extract members from a library. If no members are specified, then all are extracted.

/q
Quiet mode. No output is displayed.

LINKPIC

This program is a wrapper that runs the Microchip MPLINK linker.

SYNTAX

LINKPIC *gpath*

PARAMETERS

gpath

The generic pathname for this link. Implicit and default file names will start with [gpath]. For example, the HEX output file name will be [gpath].HEX, the map file [gpath].MAP, etc.

DESCRIPTION

Note: The environment variable MPLABDIR is assumed to be set to the directory containing the Microchip executables. If this variable is not present or empty, then the Microchip executables are assumed to be in the executables search path, in other words they can be run directly without having to specify the full path name.

The following files will be read or created/overwritten:

Table 4–7 LINKPIC input files

Filename	Description
[gpath].lkr	This is the MPLINK linker command input file. It must be present. This command file is the only way to specify object modules and libraries to link. It also contains other control information. See the Microchip MPLINK documentation for details.

Table 4–8 LINKPIC output files

Filename	Description
[gpath].out	Absolute executable created as a result of the link operation.
[gpath].map	Memory map output file.
[gpath].hex	HEX output file. This file will be in Intel INHX32 format.
[gpath].cod	Output file in Microchip format. This file is required by the Microchip debugger. Some of the other output files are derived from the .cod file after the .cod file is created.

5 The include files

5.1 STD.INS.ASPIC

This file defines a large number of macros.

This table list these macros in the same order as they are defined in STD.INS.ASPIC.

Table 5–1 STD.INS.ASPIC macros

Macro definition	Description
IREGS_DEFINE	Define all the IREG0 - IREGn interrupt routine registers. The number of these registers is set by the constant N_IREGS.
EXTERN_IREGS	Declare all the IREGn register external to this module.
GETF <adrf>	Move the contents of the file register ADRF into W. The Z flag is trashed.
GETFZ <adrf>	Move the contents of the file register ADRF into W. The Z flag is set according to the value moved.
TESTFZ <adrf>	Set the Z flag according to the contents of the file register ADRF. W is preserved.
WAITNOP NNOP	Generates code that does nothing for the next NNOP cycles. Note that this is not accurate for timing unless interrupts are disabled. This macro generates no code if NNOP is zero or less.
SKIP_WLE	Skip the next instruction if W was less than or equal to the value it was subtracted from. This assumes that the carry flag has been preserved from the last SUBWF or SUBLW instruction.
SKIP_WGT	Skip the next instruction if W was greater than the value it was subtracted from. This assumes that the carry flag has been preserved from the last SUBWF or SUBLW instruction.
SKIP_Z	Skip the next instruction if the zero flag is set.
SKIP_NZ	Skip the next instruction if the zero flag is not set.
SKIP_CARR	Skip the next instruction if a carry occurred.
SKIP_NCARR	Skip the next instruction if no carry occurred.
SKIP_BORR	Skip the next instruction if a borrow occurred.
SKIP_NBORR	Skip the next instruction if no borrow occurred.
INTR_OFF	Globally disable interrupts without changing which interrupts are individually disabled. This macro together with INTR_ON can be used around small sections of code that need to run with interrupts off. This macro works around the interrupt off bug on some processors where an interrupt can occur immediately after interrupts are disabled. This causes interrupts to be re-enabled by the RETFIE in the interrupt service routine. The work around is to verify that interrupts were indeed disabled on the next instruction and loop back if they weren't. The assembler switch INTR_OFF_BUG is set to TRUE if this processor has the bug.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
INTR_ON	Globally enable interrupts without changing which interrupts are individually enabled. This macro together with INTR_OFF can be used around small sections of code that need to run with interrupts off.
DTWORD val16	Define a 16 bit value in a table. The low byte is stored first, the high byte second.
DT32I val32	Define a 32 bit integer value in a table. The low byte is stored first, the high byte last.
LOADK32 <adr>, <32 bit constant>	Load the 32 bit constant into the memory locations starting at ADR. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the four data bytes. This macro uses CLRF instructions when possible and avoids re-loading W with the same value.
LOADK24 <adr>, <24 bit constant>	Load the 24 bit constant into the memory locations starting at ADR. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the data bytes at ADR. This macro uses CLRF instructions when possible and avoids re-loading W with the same value.
LOADK16 <adr>, <16 bit constant>	Load the 16 bit constant into the memory locations starting at ADR. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the data bytes at ADR. This macro uses CLRF instructions when possible and avoids re-loading W with the same value.
COPYN <dest>, <src>, <n>	Copy the N bytes starting at SRC into the N bytes starting at DEST. The direct register bank must be set for access to all bytes of SRC and DEST. The results are undefined if SRC and DEST overlap.
COPY32 <dest>, <src>	Copy the 32 bit value at SRC into DEST. The direct register bank must be set for access to all bytes of SRC and DEST. The results are undefined if SRC and DEST overlap.
COPY24 <dest>, <src>	Copy the 24 bit value at SRC into DEST. The direct register bank must be set for access to all bytes of SRC and DEST. The results are undefined if SRC and DEST overlap.
COPY16 <dest>, <src>	Copy the 16 bit value at SRC into DEST. The direct register bank must be set for access to all bytes of SRC and DEST. The results are undefined if SRC and DEST overlap.
SHIFT32RL1 <adr>	Perform a logical right shift of 1 bit on a 32 bit value. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the four data bytes.
SHIFT32RA1 <adr>	Perform an arithmetic right shift of 1 bit on a 32 bit value. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the four data bytes. W is trashed.
SHIFT32L1 <adr>	Perform a left shift of 1 bit on a 32 bit value. ADR is the address of the least significant byte. The remaining bytes follow at increasing memory addresses. The direct register bank must already be set for access to the four data bytes.
ADD32 <dest>, <src>, <temp>	Add the 32 bit source value into the 32 bit destination value. TEMP will be used for temporary storage, and will be trashed. The 32 bit values are stored with the low byte first. The current direct access page must be set for access to all state.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
ADD24 <dest>, <src>, <temp>	Add the 24 bit source value into the 24 bit destination value. TEMP will be used for temporary storage, and will be trashed. The 24 bit values are stored with the low byte first. The current direct access page must be set for access to all state.
ADD16 <dest>, <src>	Add the 16 bit source value into the 16 bit destination value. The 16 bit values are stored with the low byte first. The current direct access page must be set for access to all state.
SUB32 <dest>, <src>, <temp>	Subtract the 32 bit source value from the 32 bit destination value and put the result into the destination. TEMP will be used for temporary storage, and will be trashed. The 32 bit values are stored with the low byte first. The current direct access page must be set for access to all state.
SUB24 <dest>, <src>	Subtract the 24 bit source value from the 24 bit destination value and put the result into the destination. The 24 bit values are stored with the low byte first. The current direct access page must be set for access to all state.
SUB16 <dest>, <src>	Subtract the 16 bit source value from the 16 bit destination value and put the result into the destination. The 16 bit values are stored with the low byte first. The current direct access page must be set for access to all state.
NEGATE <dest>, <n>	Negate the twos complement value starting at address DEST. DEST is N bytes long and is stored with the least significant byte first. N must be greater than zero, and all bytes of DEST must be accessible with the current direct register bank selection.
FP24 <name>	Declare memory for a 24 bit floating point number. The first (least significant) byte will have the label NAME. This macro simply reserves 3 bytes, documents that these bytes are intended to be one floating point value.
FP24NEG <dest>	Negate the 24 bit floating point number at DEST. The register bank setting must be set for direct access to DEST.
FP24ABS <dest>	Set the 24 bit floating point number at DEST to its absolute value.
UART_BAUD <baud>	This macro sets assembler variables to indicate the best baud rate generator configuration for the USART in asynchronous mode, given the desired baud rate and the oscillator frequency. No code is generated. The following assembler variables are set: VAL_SPBRG - Value for the SPBRG baud rate generator register. For UARTs with 16 bit baud rate generators this will be the full 16 bit value. The low byte is then intended for SPBRG, and the high byte for SPBRGH. BAUD_REAL - Real (not desired) baud rate resulting from the selected settings. VAL_TXSTA - Value for the TXSTA register. In addition to selecting the proper baud rate generator configuration, this value also selects the following: Asynchronous mode 8 bits per character Transmitter enabled VAL_RCSTA - Value for the RCSTA register. The value will select the following setup: Serial port enabled 8 bits per character Reception enabled Address detection disabled VAL_BAUDCTL - Value for the BAUDCTL register for those processors that have this. This register is part of the enhanced USART of parts like the 18F1320. This macro produces an assembly time warning if the closest available baud rate is more then 2.9% from the desired baud rate (off by 25% of a bit time in the middle of the last bit). It produces an assembly error if the baud rate error is 5.8% or higher.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
UART_SETUP	Set up the UART according to the assembler variables VAL_SPBRG, VAL_TXSTA, VAL_RCSTA, and VAL_BAUDCTL. See the documentation for the UART_BAUD macro for a description of these variables. The values do not need to be set by the UART_BAUD macro, although that may be a convenient way to do so.
SETREG val, reg	Set the register REG to the value VAL. Both REG and VAL must be constants. The instructions are optimized. For example, CLRF is used if the value is zero.
TIMER2_USEC usec	Calculate a timer 2 setup to achieve an interrupt period of USEC microseconds. This macro generates no code, but sets the following assembler variables: TMR2_PRE - timer 2 prescaler value: 1, 4, or 16 TMR2_PER - timer 2 period divide value: 1 - 256 TMR2_POS - timer 2 postscaler value: 1 - 16 The hardware PWM period, if used, is defined by the prescaler and period values only. The number of instructions per PWM period is therefore TMR2_PRE * TMR2_PER, whereas the number of instructions per timer 2 interrupt (if enabled) is TMR2_PRE * TMR2_PER * TMR2_POS. An error is generated if the indicated period can not be attained exactly within the constraints of the timer 2 hardware. The postscaler is set to the minimum possible value that results in the specified period.
TIMER2_SETUP_INTR	Set up timer 2 to produce periodic interrupts. The timer 2 control registers are set up, and the individual timer 2 interrupt is enabled. Timer 2 divides the instruction clock by a prescaler, period, and postscaler to make the interrupt period. The following assembler variables must be previously set to define the timer 2 divider chain: TMR2_PRE - timer 2 prescaler value: 1, 4, or 16 TMR2_PER - timer 2 period divide value: 1 - 256 TMR2_POS - timer 2 postscaler value: 1 - 16 Note that the TIMER2_USEC macro can be used to compute these values given a desired interrupt period. It is an error if any of these assembler variables don't exist or are set to invalid values.
DBANK?	Invalidate the current direct register data bank assumption. This macro produces no code, only sets assembly time state. On 17Cxxx processors, this sets both the special and general register bank assumptions to unknown. These processors make no distinction between a direct or indirect bank selection.
IBANK?	Invalidate the current indirect register data bank assumption. This macro produces no code, only sets assembly time state. On 17Cxxx processors, this sets both the special and general register bank assumptions to unknown. These processors make no distinction between a direct or indirect bank selection.
UNBANK	Invalidates all register bank assumptions.
DBANKIS <adr>	Set the current direct register bank assumption to the bank containing ADR. This macro generates no code to switch the register bank. It only updates assembly time state.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
IBANKIS <adr>	Set the current indirect register bank assumption to the bank containing ADR. This macro generates no code to switch the register bank. It only updates assembly time state.
RBANKIS <adr>	Set the current special function register bank assumption to the bank containing ADR.
GBANKIS <adr>	Set the current general RAM register bank assumption to the bank containing ADR.
RBANK?	Invalidates the current special function register bank assumption.
GBANK?	Invalidates the current general RAM register bank assumption.
DBANKIF <adr>	Set the register bank for direct access to address ADR. This macro sets the RP0, RP1 bits of STATUS appropriately. The bank bits are not set if they are assumed to already be set correctly as indicated by the assembler variable CURRDB. CURRDB is updated to the new setting. On 18 family processors, this macro sets BSR. No code is emitted if and the bank setting is not altered if ADR is within the access bank, and can therefore be accessed regardless of the BSR setting.
DBANK <adr>	Unconditionally set the direct access register bank for access to address ADR. The assembler state is updated.
IBANKIF <adr>	Set the register bank for indirect access to address ADR, if needed. The assembler variable CURRIB is assumed to indicate the current indirect register bank setting. CURRIB is updated. This macro sets the IRP bit of STATUS appropriately.
IBANK <adr>	Unconditionally set the indirect access register bank for access to address ADR. CURRIB is updated.
RBANKIF <adr>	This macro is only defined for processors that have different bank settings for the built in special function registers and the general RAM registers. Set the special function register bank for access to ADR. Nothing is done if ADR is not within the special banked special function register range.
GBANKIF <adr>	This macro is only defined for processors that have different bank settings for the built in special function registers and the general RAM registers. Set the general RAM register bank for access to ADR. Nothing is done if ADR is not within the general RAM register range.
DEFRAM <address>	Set up for allocating RAM with subsequent RES directives. ADDRESS is the address that will be passed to DBANKIF to access any of the RAM defined in this section. If ADDRESS indicates normal banked memory, then the appropriate .BANKn linker section will be started with a UDATA directive. If ADDRESS is in common RAM, then the RAM will be in the .UDATA_SHR section. If ADDRESS is in the access bank of an 18 family, then the RAM will be allocated in the .UDATA_ACS section. If the new linker section would be the same as the linker section defined by the previous DEFRAM, then no new directives are written since only one occurrence of a linker section is allowed per source module. The assembler symbol DEFRAM_LAST is used to track linker section of the last DEFRAM invocation. This symbol has the following values: -1 - No previous DEFRAM section -2 - Last linker section was .UDATA_SHR -3 - Last linker section was .UDATA_ACS 0-N - Last linker section was .BANKn

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
PUSHREG reg	Push the contents of the register REG onto the stack. The direct bank must already be set for access to REG. On some processors, it is more efficient to push a set of general registers onto the stack with one PUSHREGS macro than to use the PUSHREG macro several times. This macro also trashes W on some processors. In that case the assembler variable W_TRASHED is set to TRUE, otherwise it is left unaltered. Some of the specifics per processor are: 16 family - Relatively inefficient to push a single value. W trashed. 18 family - Efficient to push a single value. Multiple PUSHREG just as efficient as a single PUSHREGS.
POPREG reg	Pop the top of the software stack into REG. The direct bank must already be set for access to REG. On some processors, it is more efficient to pop a set of general registers onto the stack with one POPREGS macro than to use the POPREG macro several times. This macro also trashes W on some processors. In that case the assembler variable W_TRASHED is set to TRUE, otherwise it is left unaltered. Some of the specifics per processor are: 16 family - Relatively inefficient to pop a single value. W trashed. 18 family - Efficient to pop a single value. Multiple POPREG just as efficient as a single POPREGS.
PUSHREGS <regflags>	Push the indicated general registers onto the stack. The REGFLAGS argument is the OR of the REGFn flags for all the registers that are to be pushed. The registers are pushed in low to high order. It is more efficient to call this macro once with all the registers to push than to call it separately for each register. NOTE: The current indirect bank assumption must be correct. If the current indirect bank setting is not known, then the IBANK? macro should be called before this macro. Trashed: W, FSR, indirect register bank with CURRIB updated.
POPREGS <regflags>	This macro performs the reverse operation to the PUSHREGS macro. The REGFLAGS argument is the OR of a set of REGFn flags to indicate the registers to be popped from the stack. The registers are popped in high to low order. It is more efficient to call this macro once with all the registers to pop than to call it separately for each register. NOTE: The current indirect bank assumption must be correct. If the current indirect bank setting is not known, then the IBANK? macro should be called before this macro. Trashed: W, FSR, indirect register bank, CURRIB updated
ENTER <regflags>	This macro performs the "standard" operations on subroutine entry. These actions are: 1 - Invalidate the current register bank assumptions. 2 - Save the registers indicated by REGFLAGS onto the software stack. REGFLAGS is the or of the REGFn flags for each register to save. REGFLAGS may be NOREGS to indicate no registers are to be saved. 3 - Sets SAVEDREGS to the flags indicating which registers were saved. This can be used to automatically restore the saved registers later.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
LEAVE <regflags>	This macro performs the "standard" operations on subroutine exit. These actions are: 1 - Restore the registers indicated by REGFLAGS from the software stack. They are assumed to have been pushed in low to high register number order, and will be popped in high to low order. REGFLAGS may be NOREGS to indicate no registers are to be restored. 2 - Return from the subroutine. 3 - Invalidate the current register bank assumptions. Note that this sets assembly state, not runtime state. The assumptions are therefore propagated in source code, not execution order. NOTE: The current indirect register bank assumption must be correct when this macro is called. Note that it is set correctly by the ENTER macro, but may have been altered by intervening code. Use the IBANK? macro to invalidate the current assumption if not sure.
LEAVEREST	Just like LEAVE except that the registers indicated by SAVEDREGS are automatically restored. SAVEDREGS should have been set by the ENTER macro to the routine being left.
GLBSUB <name>, <regflags>	Declare the start of a new global subroutine. <name> will be declared as the global subroutine entry point label at the current location. The subroutine entry point will be immediately followed by ENTER <regflags>. In other words, the new subroutine will be start by pushing the registers identified by <regflags> onto the stack. Nothing will be pushed and no code generated if <regflags> has the value NOREGS.
LOCSUB <name>, <regflags>	Just like GLBSUB except that the subroutine will not be global. It will only be known within its module.
GLBENT <name>	Declare a global entry point that can be jumped to from other modules.
LOCENT <name>	Declare a local entry point that can only be jumped to from within the same module.
SETPAGE <address>	Select the code page (by setting PCLATH) containing the indicated address.
MYPAGE	Sets PCLATH to the current code page, if this machine has multiple code pages. W may be trashed.
GCALL <subroutine>	Call a global subroutine, which could be anywhere in program memory. W is trashed between the caller and the subroutine, and can therefore not be used to pass data to the subroutine. The assumed register banks are set to invalid after the subroutine returns. No code page selection code is generated on machines that only have one code page.
GCALLNR <subroutine>	Just like GCALL except that the current program memory page is not restored after the subroutine returns. This saves unnecessary instructions if the current code page setting is not used by subsequent code. This could be the case, for example, if another global subroutine was called immediately afterwards. WARNING: This macro will cause subsequent code to fail if it does rely on the current program memory page setting. Local CALLs and GOTOs rely on the current page setting.
MCALL <subroutine>	Call a local subroutine within the same module (code section, actually). The subroutine is therefore guaranteed to be on the same code page. The assumed register banks are invalidated after the subroutine returns.
GCALLWR <subroutine>	Just like GCALL, except that W is preserved on return from the subroutine. Note that W is still trashed from the caller to the subroutine.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
GCALLR <subroutine> <putret>	Just like GCALL, except that the subroutine is assumed to return a value in W. This return value will be stored in PUTRET. Note that PUTRET must be in global (any bank) memory unless the subroutine is known to explicitly set the direct register bank. W is trashed. The direct and indirect register bank settings are preserved from the subroutine.
MCALLR <subroutine> <putret>	Just like MCALL, except that the subroutine is assumed to return a value in W. This return value will be stored in PUTRET. Note that PUTRET must be in global (any bank) memory unless the subroutine is known to explicitly set the direct register bank. W is trashed. The direct and indirect register bank settings are preserved from the subroutine.
GJUMP <program address>	Jump to a location that could be anywhere in program memory. W is trashed before execution starts at the new location. The assumed register banks are set to invalid for source code immediately following the GJUMP.
FLAGS_DEFINE	Define the GFLx variables required to hold all the flags. This macro will create one RES directive for each flag variable and declare it GLOBAL. This macro must only be called after all the /FLAG directives, else NFLAGB will not be valid.
FLAGS_CLEAR	Clear all the global flags defined with the FLAG macro. The GFLx variables are assumed to be declared in the GBANK register bank, and NFLAGB is the number of GFLx variables.
EXTERN_FLAGS	Declare all the GFLx flag bytes as external.
FIFO_DEFINE name size	Define a first in, first out queue. The symbol NAME will be defined as the first byte of the queue structure. Size is the maximum number of data bytes the queue will be able to hold.
FIFO_INIT name	Initialize the FIFO at NAME. The register bank must be set for access to the FIFO state.
FIFO_SKIP_EMPTY name	Skips the next instruction after the macro if the FIFO at NAME is empty. The register bank must be set for access to the FIFO state.
FIFO_SKIP_NEMPTY name	Skips the next instruction after the macro if the FIFO at NAME contains at least one data byte. The register bank must be set for access to the FIFO state. W may be trashed.
FIFO_SKIP_FULL name size	Skips the next instruction after the macro if the FIFO at NAME is completely full. The register bank must be set for access to the FIFO state. W is trashed.
FIFO_SKIP_NFULL name size	Skips the next instruction after the macro if the FIFO at NAME is not completely full. The register bank must be set for access to the FIFO state. W is trashed.

Table 5–1 (Cont.) STD.INS.ASPIC macros

Macro definition	Description
FIFO_PUT name size data	Add the byte in DATA as the last byte in the FIFO at NAME. SIZE must be the maximum number of bytes the FIFO was defined to hold. The register bank must be set for access to the FIFO and DATA. Since the FIFO is usually not in global RAM, this means DATA must be either in global RAM or in the same register bank as the FIFO. The FIFO may be trashed if it is already full. This should be checked before this macro is called. Note that if the FIFO could be accessed from the interrupt service routine, then interrupts should be temporarily disabled around this macro. The indirect register bank must be set for access to the FIFO. W is trashed.
FIFO_GET name size data	Get the next byte from the FIFO at NAME into DATA. SIZE must be the maximum number of bytes the FIFO was defined to hold. The register bank must be set for access to the FIFO and DATA. Since the FIFO is usually not in global RAM, this means DATA must be either in global RAM or in the same register bank as the FIFO. The FIFO may be trashed if it is already empty. This should be checked before this macro is called. Note that if the FIFO could be accessed from the interrupt service routine, then interrupts should be temporarily disabled around this macro. The indirect register bank must be set for access to the FIFO. W is trashed.

Tips-n-Tricks

This part of the document provides general information that does not nesessarily originates from the web pages of Embedinc.

6 Creating a new project

Contributor : Jan-Erik Söderholm, S:T Anna Data, Sweden

This was the steps I did when creating my first (rather simple) project in the Embedinc development environment. I started with a fresh install of the environment, no HOS or HAL kits installed. The project is for 18F1320 using the UART, interupt from multiple sources, PWM (to create a 38KHz IR signal) and some other features.

Since the development environment expects 3 character project names, I selected **D01** for this specific project.

6.1 Creating the directory for the projects source files

This is realy quite easy, just create a sub-directory to the **\embedinc\source** directory named the same as the selected project name. In my case **\embedinc\source\D01**.

6.2 Creating the directory for the projects output files

This is just as easy, just create a sub-directory to the **\embedinc\src** directory named the same as the selected project name. In my case **\embedinc\src\D01**.

6.3 Copy files from the PIC directory

Now the file needed for the project can be copied from the PIC directory. These file are "skelleton" files that in some cases need to be modified a bit to suite the actual project. The files I copied was :

Table 6–1 Files copied from the \PIC directory

\source\PIC file	\source\D01 File	Note
p18f1220.linkpic	D01.linkpic	The control file for MPLINK. In this project, I saw no reason to change anything in the linkpic file, and I can't really say I understand it anyway...
qqqlib.ins.aspic	D01lib.ins.aspic	This module sets up the environment (such as processor type and osc frequency) for your project. Has to be edited.
qqq.ins.anspic	D01.ins.aspic	Stuff like **/FLAG**, **/INBIT** and **/OUTBUT** definitions. Has to be edited.

Table 6–1 (Cont.) Files copied from the \PIC directory

\source\PIC file	\source\D01 File	Note
qqq_strt.aspic	D01_strt.aspic	This is the "entry" module in the project. Update the _CONFIGx symbol(s) and don't forget to edit the include at the top (true for most of the files, b.t.w) ! This **STRT** module calles the following routines • **stack_init**, found in **stack.ins.aspic**. (If the project doesn't use any "software stack", this call could be commented out.) • **port_init**, found in **qqq_port.aspic**. • **uart_init**, found in **qqq_uart.aspic**. • **cmd_init**, found in **qqq_cmd.aspic**. • **intr_init**, found in **qqq_intr.aspic** or **qqq_intr18.aspic**. These calls can be commented out if the specific feature isn't used. The **STRT** module ends with a jump to the **INIT** module.
qqq_init.aspic	D01_init.aspic	Here you put executable code that is project specific and that should run att startup of the project (power on). The **INIT** module end with a jump to the **MAIN** module.
qqq_main.aspic	D01_main.aspic	The **MAIN** mudule, as it is, is built as a kind of "state machine". It checkes a number of global flags (probably set by an interupt routine or some of the "handlers"), and if there is something to do, jumps of to the specific routine (that should end with a **gjump loop_main** command to jump back to the main loop). I suppose this could be changed into some other logic but I decided to use it as it is for this specific project.
qqq_intr18.aspic	D01_intr.aspic	This module contains both the **INTR_INIT** routine (called from the **STRT** module once at startup) and the interupt service routine that will be placed at the right address (the interupt vector address). As it is, the ISR contains code to save and restore context and check for two interupt sources, the "UART receive interrupt" and the "UART transmitter ready interrupt". Other checks has to be added as needed. Note that for a 16-series PIC, use **qqq_intr.aspic** module ! The qqq_intr18.aspic module also has code to deal with the two interrupt priorities on the 18-series PICs.
qqq_port.aspic	D01_port.aspic	Nothing to change. Contains the **port_init** routine.
qqq_uart.aspic	D01_uart.aspic	Change any project specific stuff, such as UART speed. Contains a number of different routines to send and receive data over the UART line.

Apart from the files listed in the table above, the project can have additional modules called **D01_xxxx.aspic**, that can be added as needed.

6.4 Creating the build script

The build script is a DOS BAT file that runs all Embedinc tools to build the project from the source project directory. Since there is an entry in the PATH to the **\embedinc\COM** directory (setup while installing the environment), an easy way is to put the build script there.

I created **build_D01.bat** with the following contents :

Example 6–1 Example of project build file

```
@echo off
rem
rem   BUILD_D01.BAT
rem
rem   Build the D01 firmware from the D01 library.
rem

call src_ins_aspic D01 D01lib
call src_ins_aspic D01 D01

call src_aspic D01 D01_cmd
call src_aspic D01 D01_init
call src_aspic D01 D01_intr
call src_aspic D01 D01_main
call src_aspic D01 D01_port
call src_aspic D01 D01_uart
call src_aspic D01 D01_strt

call src_libpic D01 D01
call src_expic D01 D01
```

When using a build script like the one above (using src_libpic), a file called **D01.mlist** has to be created. It's just a plain text file with one line for each object file in the project. In this example it should look like :

Example 6–2 Example of mlist file

```
D01_cmd.o
D01_init.o
D01_intr.o
D01_main.o
D01_port.o
D01_uart.o
D01_strt.o
```

7 Using UltraEdit with the environment

Contributor : Jan-Erik Söderholm, S:T Anna Data, Sweden

UltraEdit ("UE") is a full function text editor that is highly configurable. In this section I'll try to describe the steps I'v done to get a working environment in UE.

7.1 Download and Install

Easy, go to www.ultraedit.com and download a kit.

I don't remember doing anything special when installing...

7.2 Language highlightning

The "words" to highlight in UE are in the the C:\Program\UltraEdit\WORDFILE.TXT file. The part of the file dealing with PIC asm (or aspic) files looks like the example below. This file should be insterted in the WORD_FILE.TXT file as "language number 10".

Note: I have not created this file myself, but have forgotten where I got it. And I have changed it a bit, so it works better with the PIC18-series PICs.

Don't expect this file/example to be complete. And some of the keywords don't work (for me), such as "tblrd*", probably because "*" is also in the "Delimiters" section.

Contact the author for a up-to-date copy of this file.

Example 7–1 Part of WORDFILE.TXT for PIC support

Example 7–1 Cont'd on next page

Example 7–1 (Cont.) Part of WORDFILE.TXT for PIC support

```
/L10"Microchip PIC Asm" Nocase Line Comment = ; File Extensions = ASM ASPIC INC
/Delimiters = ~!@$%^&*()-+=|\{}[]:;"'<> ,   .
/Function String = "%[a-zA-Z_]*"
/C1 ASSEMBLER DIRECTIVES
#DEFINE #INCLUDE #UNDEFINE
BANKISEL BANKSEL
CBLOCK CODE CONSTANT
DATA DB DE DT DW
ELSE END ENDC ENDIF ENDM ENDW EQU ERROR ERRORLEVEL EXITM EXPAND EXTERN
FILL
GLOBAL
IDATA IF IFDEF IFNDEF
LIST LOCAL
MACRO MESSG
NOEXPAND NOLIST
ORG
PAGE PAGESEL PROCESSOR
RADIX RES
SET SPACE SUBTITLE
TITLE
UDATA UDATA_OVR UDATA_SHR
VARIABLE
WHILE
__BADRAM __CONFIG __IDLOCS __MAXRAM
=
/C2 PIC16CXX INSTRUCTION SET
addlw addwf addwfc andlw andwf
bcf bsf btg btfsc btfss
call clrf clrw clrwdt comf
decf decfsz dcfsnz daw
goto
clrf
incf incfsz infsnz iorlw iorwf
movf movlw movwf movff
mullw mulwf negf
nop
rlcf rlncf rrcf rrncf
retfie retlw return rlf rrf
sleep sublw subwf swapf setf subwfb
cpfseq cpfsgt cpfslt
xorlw xorwf
tblrd* tblrd*+ tblrd*- tblrd+* tblwt* tblwt*+ tblwt*- tblwt+*
tstfsz
/C4 PRE-DEFINED REGISTER LABELS
ADCON0 ADCON1 ADCON2 ADRES
CCP1CON CCP2CON CCPR1H CCPR1L CCPR2H CCPR2L CMCON
EEADR EECON1 EECON2 EEDATA
F FSR
GPIO
INDF INTCON IOC IOCA
LCDCON LCDD00 LCDD01 LCDD02 LCDD03 LCDD04 LCDD05 LCDD06 LCDD07
LCDD08 LCDD09 LCDD10 LCDD11 LCDD12 LCDD13 LCDD14 LCDD15 LCDPS LCDSE
OSCCAL OPTION_REG
PCL PCLATH PCON PIE1 PIE2 PIR1 PIR2 PORTA PORTB PORTC PORTD PORTE
PORTF PORTG PR2 RCREG RCSTA RTCC
SPBRG SSPADD SSPBUF SSPCON SSPSTAT STATUS
T1CON T2CON TMR0 TMR1H TMR1L TMR2 TRISA TRISB TRISC TRISD
TRISE TRISF TRISG TXREG TXSTA
VRCON
W WPU WPUA WREG
/C5 PRE-DEFINED BIT LABELS
ADCS0 ADCS1 ADIE ADIF ADON
BF BO BRGH
C C1OUT C2OUT
```

Example 7–1 Cont'd on next page

Example 7–1 (Cont.) Part of WORDFILE.TXT for PIC support

```
CAL0 CAL1 CAL2 CAL3 CAL4 CAL5
CCP1IE CCP1IF CCP1M0 CCP1M1 CCP1M2 CCP1M3 CCP1X CCP1Y
CCP2IE CCP2IF CCP2M0 CCP2M1 CCP2M2 CCP2M3 CCP2X CCP2Y
CHS0 CHS1 CHS2
CKE CKP CM0 CM1 CM2 CIS CMIE CMIF CREN CS0 CS1 CSRC
DA DC
EEIE EEIF
FERR
GIE GO_DONE GPIE GPIF
IBF IBOV INTE INTEDG INTF IRP
LCDEN LCDIE LCDIF LMUX0 LMUX1 LP0 LP1 LP2 LP3
NOT_BOD NOT_GPPU NOT_PD NOT_POR NOT_RBPU NOT_RBWU NOT_TO NOT_T1SYNC
OBF OERR
P PA0 PA1 PCFG0 PCFG1 PCFG2 PEIE POR PS0 PS1 PS2 PSA PSPIE PSPIF PSPMODE
RBIE RBIF RBWUF RCIE RCIF RD
RP0 RP1 RW RX9 RX9D
S SE0 SE5 SE9 SE12 SE16 SE20 SE27 SE29
SLPEN SMP SPEN SREN SSPEN SSPIE SSPIF SSPM0 SSPM1 SSPM2 SSPM3 SSPOV SYNC
T0CS T0IE T0IF T0SE
T1CKPS0 T1CKPS1 T1IE T1IF T1OSCEN
T2CKPS0 T2CKPS1
TMR1CS TMR1GE TMR1IE TMR1IF TMR1ON TMR2IE TMR2IF TMR2ON
TOUTPS3 TOUTPS2 TOUTPS1 TOUTPS0
TRISE0 TRISE1 TRISE2 TRMT
TX89 TX9 TX9D TXD8 TXEN TXIE TXIF
UA
VGEN VR0 VR1 VR2 VR3 VREN VROE VRR
WCOL WR WREN WRERR
Z
/C6 SYMBOLS
+
,
-
/
<
>
/C7 MACRO EQUATES
ADDCF ADDDCF
B BC BNC BNDC BNZ BZ
COPYN COPY32 COPY24 COPY16 CLRC CLRDC CLRZ
LCALL LGOTO
MOVFW
NEGF
SETC SETDC SETZ SKPC SKPDC SKPNC SKPNDC SKPNZ SKPZ SUBCF SUBDCF
TSTF
/C8 OLINS MACRO OPERATIONS
ADD32 ADD24 ADD16
BANKADR BANKADRG BANKADRR BANKOF
DBANK DBANK? DBANKIF DBANKIS DEFRAM DTWORD DT32I
ENTER EXTERN_FLAGS EXTERN_IREGS
FP24 FP24ABS FP24NEG FREQ_INST FREQ_OSC FLAGS_CLEAR FLAGS_DEFINE
FIFO_DEFINE FIFO_INIT FIFO_SKIP_EMPTY FIFO_SKIP_NEMPTY FIFO_SKIP_FULL
FIFO_SKIP_NFULL FIFO_PUT FIFO_GET
GBANK? GBANKIF GBANKIS GCALL GCALLNR GCALLR GCALLWR GETF GETFZ
GJUMP GLBENT GLBSUB
IBANK IBANK? IBANKADR IBANKIF IBANKIS IBANKOF INACCESSBANK INBANKED
INTR_OFF INTR_ON IREGS_DEFINE
JUMP
LEAVE LEAVEREST LOADK32 LOADK24 LOADK16 LOCENT LOCSUB
MCALL MCALLR MYPAGE
NEGATE N_IREGS NSEC_INST
POPREG POPREGS PUSHREG PUSHREGS
RBANK? RBANKIF RBANKIS REG
SETPAGE SETREG SHIFT32RL1 SHIFT32RA1 SHIFT32L1
```

Example 7–1 Cont'd on next page

Example 7–1 (Cont.) Part of WORDFILE.TXT for PIC support

```
SKIP_FLT SKIP_FLE SKIP_FEQ SKIP_FGT SKIP_FGE SKIP_FNE SKIP_ERR SKIP_NERR
SKIP_WLE SKIP_WGT SKIP_Z SKIP_NZ SKIP_CARR SKIP_NCARR SKIP_BORR SKIP_NBORR
SUB32 SUB24 SUB16
TESTFZ TMR0_PER TMR0_PRE TMR0_PSA TIMER0_SETUP_INTR TIMER0_USEC
TMR1_PER TMR1_POS TMR1_PRE TIMER1_SETUP_INTR TIMER1_USEC
TMR2_PER TMR2_POS TMR2_PRE TIMER2_SETUP_INTR TIMER2_USEC
UART_BAUD UART_SETUP UNBANK
VAL_BAUDCTL VAL_RCSTA VAL_SPBRG VAL_TRIS VAL_TXSTA VAL_PORT
W_TRASHED WAITNOP
```

7.3 Adding the build script to the UE menus

It's possible to create a toolbar button to directly run the build script for the project. Something like this :

1 If not already done, create a new project in UE. Project -> New-project

  This will create a .prj file

2 Now select Advanced -> Project Tool Configuration.

  Enter :

  - A command line to execute the build script.
  - A working directory.

    Enter \embedinc\source\your-proj directory.
  - Enter a Menu Item Name.
  - Select "Output to list box".
  - Select "Capture Output".
  - Select "Save all file first"

3 Click "Insert" and "Exit".

Now the new menu command should be available in the Advanced menu.

Right-click on the toolbar, select "Customize" and drag the menu item onto the toolbar. (Well, try it, it's easier to do then to describe).

I have also in the same way created a button to run Xwisp/Wisp628 directly via a button on the toolbar.

8 Using a MAKE utility to speed up the builds

The Embedinc development environment, as-is, always rebuilds all modules. The rebuild can run faster if using a MAKE tool to selectivily only rebuild those modules needed. Below is an example of a modified build script that uses the MAKE tool from GnuWin.

Note: I had some problems when calling the Embedinc "wrappers" (the src_*.bat files) from the makefile. So I changed to calling the preprocessors directly.

Example 8–1 Build script using make

```
@echo off
rem
rem   BUILD_D01
rem
rem   Build the D01 firmware from the D01 library.
rem   Using
rem

cd c:\embedinc\src\D01
copy ..\..\source\D01\*.aspic *.*
rem copy ..\..\source\D01\*.mlist *.*
copy ..\..\source\D01\*.lkr *.*
copy ..\..\source\D01\makefile *.*

make -i -r
```

A makefile file used in this case looks like this :

Example 8–2 makefile to be used with the build script

```
prepic          =  c:\embedinc\com\prepic
asm             =  c:\Program\MPLAB IDE\MCHIP_Tools\MPASMWIN.EXE
link            =  c:\Program\MPLAB IDE\MCHIP_Tools\MPLINK.EXE
D01_objs        =  d01_strt.o d01_init.o d01_cmd.o d01_intr.o d01_main.o \
                   d01_port.o d01_s16.o d01_uart.o d01_tmr.o  d01_enc.o  \
                   d01_pwm.o

%.asm   : %.aspic
 $(prepic) $< $@

%.inc   : %.ins.aspic
 $(prepic) $< $@

%.o     : %.asm
 $(asm) /c- /q /o$@  $<

d01.hex : $(D01_objs)
 echo "FILES $(D01_objs)" > temp.lkr
 $(link) D01.lkr temp.lkr /m d01.map /o $@

d01_strt.o  : d01.inc d01lib.inc

d01_init.o  : d01.inc d01lib.inc

d01_cmd.o   : d01.inc d01lib.inc

d01_intr.o  : d01.inc d01lib.inc

d01_main.o  : d01.inc d01lib.inc

d01_port.o  : d01.inc d01lib.inc

d01_s16.o   : d01.inc d01lib.inc

d01_uart.o  : d01.inc d01lib.inc

d01_tmr.o   : d01.inc d01lib.inc

d01_enc.o   : d01.inc d01lib.inc

d01_pwm.o   : d01.inc d01lib.inc
```

9 Using the environment with PIC12F629/675

Contributor : Jan-Erik Söderholm, S:T Anna Data, Sweden

Note: I don't claim to know everything (not even "a lot") about the linker and link files. This is just what I have found out after a few hours experimenting. I'm positive that when Olin has looked over this, it will look completely different :-)

There is one feature of the 12F629/675 PICs that differs from any other PIC. *All* GPR memory is shared between the two memory banks (bank0 and bank1). The current link file (see Example 9–1) doesn't really handle this fact in the best way. There are still two separate memory areas, the *globalram* and the *bank0* memory areas, and you have to "spread" your variables between these two areas even if both areas are equivalent from a programming standpoint. So, you could just as well put all variables into the *globalram* area.

Example 9–1 Original P12F629.linkpic file

```
//   Linker control file for the PIC 12F629 processor.
//
CODEPAGE NAME=config START=0x2007 END=0x2007 PROTECTED //configuration word
CODEPAGE NAME=idlocs START=0x2000 END=0X2003 PROTECTED //ID words
CODEPAGE NAME=eedata START=0x2100 END=0x217F PROTECTED //Initial EEPROM data

CODEPAGE NAME=code0      START=0  END=0x3FE //code page 0

SECTION NAME=.config ROM=config
SECTION NAME=.IDLOCS ROM=idlocs
SECTION NAME=.EEDATA ROM=eedata

SHAREBANK NAME=globalram  START=0x20  END=0x2F PROTECTED //UDATA_SHR area of bank 0
SHAREBANK NAME=globalram  START=0xA0  END=0xAF PROTECTED //aliased from bank 1

DATABANK NAME=bank0  START=0x30  END=0x5F //register bank 0

SECTION NAME=.udata_shr RAM=globalram  //global memory mapped to all pages
SECTION NAME=.BANK0 RAM=bank0          //for registers explicitly in bank 0
```

In the original linkfile, *bank0* area is larger then the *globalram* area. I changed the linkfile and moved most of the GPRs into globalram. I could not delete the bank0 completely, since I got a couple of errors I could not investigate further. Anyway, my changes to the linkfile are shown in Example 9–2.

Example 9–2 My modified P12F629.linkpic file

```
   .
   .
   .
SHAREBANK NAME=globalram  START=0x20  END=0x4F PROTECTED //UDATA_SHR area of bank 0 ❶
SHAREBANK NAME=globalram  START=0xA0  END=0xCF PROTECTED //aliased from bank 1 ❷

DATABANK NAME=bank0  START=0x50  END=0x5F //register bank 0 ❸
   .
   .
   .
```

❶ The globalram area is made larger (from h20-h2F to h20-h4F).

❷ The same for the bank1 aliased area.

❸ The bank0 is made smaller.

The actual size of the *globalram* area could of course be made into any size. I'm not sure of what standard modules allocates memory from the *bank0* area, so I don't know how small it can be. Anyway, now *all* allocation of GPRs can be done using *defram gbankadr* as in Example 9–3.

Example 9–3 Allocating GPRs

```
   .
   .
   .
            defram  gbankadr ❶

my_va1      res 1 ❷
            global  my_var1 ❸

my_va2      res 1 ❹
            global  my_var2 ❺
   .
   .
   .
```

❶ Set the ram area to *gbankadr*.

❷ The *RES* directive allocates from the *globalram*.

❸ And make it global (if needed).

❹ A second variable

❺ also made global

Now, this all works just fine, but of course, it would have been even better if *all* GPRs could have been handled in the same way, as *globalram*. But that was beyond my understanding of link files.

I have checked that *dbankif* will correctly handle the globalram. There will be no "set" or "clear" of bank bits when "switching" from *any* SFR to a GPR, which is correctly.