Pure C English Overview

Pure C English Overview

(or, "What Your 'Mutter' Never Told You About Pure C")

by Dan Wilga

Table of Content

Pure C English Overview 1
Table of Content 1
Installation 2
The Pure C Environment (PC.PRG) 3
Editing Keys 3
Error Message Windows 3
Pure C's Help System 4
Help Window Controls 4
The Item Selector 4
Modifications of Menu Commands 5
Compiler Options 5
Preprocessor 6
Assembler Options 7
Linker Options 7
Project (.PRJ) Files 7
The Runtime Startup Modules 8
Writing Desk Accessories 9
Helpful Hints 9
Using CFG and PRJ files: 9
Using warning level 1: 9
Pre-defined Data Types 10
The Help Compiler 10
The Pure Debugger 12
Pure Assembler 13
Assembly Language Considerations: Parameter Passing 18
Assembly Language Considerations: Return Values 19

Pure C English Overview

The Pure C package comes, as you have probably noticed by this time, with German documentation. As this is not likely to change in the near future, we have decided to present people who purchase the package from us with a description of the important features of the programs in English.

Certainly, the information in this document is far from the content of the full manuals. However, careful attention has been paid to covering those details which would not normally be obvious without understanding the manuals.

We strongly suggest that you read as much of this document as possible. While you may never use some of the information covered, many important features of the package may be revealed to you by reading it fully.

Installation

Probably the easiest way to install Pure C is in the format in which it is provided. Simply create a PURE_C folder somewhere on one of your hard drives (with about 2 Mb free) and copy the contents of all three disks into it.

This is a description of the major files in the distribution:

PC.PRG	Programming environment
PC.CFG	Default configuration file
CPP.TTP	Pure C compiler pre-processor
PCC.TTP	Pure C compiler
PCC.CFG	Pure C compiler default configuration
PD.PRG	Pure Debugger
PD.CFG	Default configuration file
BGIOBJ.TTP	Converts BGI fonts to .O files
DISPOBJ.TTP	Display object files
HC.TTP	Help compiler
PASM.TTP	Pure Assembler
PLINK.TTP	Object file linker
C.HLP	C language help
LIB.HLP	C libraries help
PASM.HLP	Pure Assembler help
PD.HLP	Pure Debugger help
FONTS\	Fonts used by BGI routines
PC_FSEL\	Replacement file selector
PC_HELP\	Help desk accessory
INCLUDE\	Header files
LIB\	

PC881LIB.LIB	68881/2 floating point library
PCBGILIB.LIB	Borland Graphics Interface library
PCEXTLIB.LIB	Pure C extensions library (see: ext.h)
PCFLTLIB.LIB	Floating point library
PCGEMLIB.LIB	AES/VDI library
PCLNALIB.LIB	Line A library
PCSTDLIB.LIB	Standard library (stdio, etc.)
PCTOSLIB.LIB	BIOS/XBIOS/GEMDOS library
PCSTART.S	Source for default runtime module
PCSTART.O	Object file for same

PCVSTART.S	Source for runtime module with extended argument (ARGV) passing
PCVSTART.O	Object file for same
PCXSTART.S	Source for runtime module with i/o redirection as part of command line
PCXSTART.O	Object file for same

The Pure C Environment (PC.PRG)

While the separate parts of the Pure C compiler/assembler/debugger can all be used independently, the easiest way to manage a large project is by using the integrated environment, PC.PRG.

Editing Keys

Shift up arrow	Scroll up one page
Shift down arrow	Scroll down one page
Shift left arrow	Move to start of line
Shift right arrow	Move to end of line
Control left arrow	Move to start of word
Control right arrow	Move to start of next word
Backspace	Delete character to left of cursor
Delete	Delete character to right of cursor
Home	Start of file
Shift Home	End of file
Control	Y Yank (cut) current line
Click	Move cursor
Shift click	Add to selection
Double-click	Select word
Shift double-click	Select an entire line
Undo	Undo last keystroke
Insert	Same as Paste

Note that using the scroll bar does not affect the cursor location. This means that if you use the scroll bar to reposition the file being edited and then type a character, that character will appear where the cursor was (and still is) before the window was scrolled.

If a block is selected when a normal key is pressed on the keyboard, that entire block will be replaced with the one character, so caution should be used.

Error Message Windows

A second type of window, which cannot be edited, appears when the compiler generates error messages. The contents of this window can be viewed like other text windows, but they cannot be changed. To quickly jump to a particular error in the appropriate source file, simply double-click anywhere on an error message. This action is similar to the "Find Error" menu entry.

Pure C's Help System

Yet another type of window which can be opened is for help messages. There are two ways to access the help, either directly from the menu or by pressing the Help key. In the case of the Help key, the help libraries will be searched for an entry which matches the word currently under the cursor. This can be especially helpful for situations where you cannot remember what parameters to pass to a library function.

Help Window Controls

• Double-click on an underlined word or phrase	Jump to that help reference
• Help key while cursor is at underlined word	Jump to that help reference
• Undo key	Back up to previous help screen

Another use of the help system is to point out the declaration of a variable, constant, or function **within your own source code**. For instance, if you are working on a program which uses a number of header files to define variables and you want to find out how the variable "MaxValue" is defined, you simply have to place the edit cursor on the word "MaxValue" somewhere in the source code and press the Help key. If you have just recently compiled your program (and Pure C's internal caches have not been flushed) then the header file in which "MaxValue" is declared will be opened and the appropriate line will be shown. Note that this feature cannot work if you quit Pure C and then re-run it, because the caches are flushed; you must re-compile the program.

The Item Selector

Pure C employs (yet another!) enhanced item selector which offers many advantages to the standard Atari item selector. All of its operations can be controlled from the keyboard:

• Alt <drive letter>	Select disk drive
• Up/Down arrows	Change selected file/folder
• Return	Open selected folder/file
• Insert	New filename
• Delete	Delete selected file
• Esc	Swap disks (update file list)
• Undo	Back up one directory level
• Control-U	Exit item selector (like Cancel button)
• Extensions (*.C, *.H, etc.)	See page 15 of your Pure C Compiler manual

One other nice feature of the item selector is the ability to select a file or folder from a long list by simply typing the first few characters of its name.

For instance, suppose you have the following items:

```
*BIN
*LIB
 BAR.C
 FOO.C
 FOX.C
 ZZZ.C
```

To open the LIB folder, you merely have to press 'L' (select the folder) and then hit the Return key (Open it). To open the file FOX.C, in this case, you can either press 'F' and then move the selection down one line with the down arrow key, or you can type 'OX' to narrow the search. In either case, pressing Return will cause the file FOX.C to be opened.

Pure C English Overview

Modifications of Menu Commands

Two menu commands have alternate forms which are not otherwise obvious:

- Control Shift Q — Quit without autosaving configuration
- Shift Alt D — Debug, removing PC.PRG from memory first

Compiler Options

Some of the compiler options are explained in more detail here. The remainder can be found in the "Compiler..." dropdown menu entry.

-2	Generate 68020 code. Resulting program will not run on a 68000 CPU!
-**8**	Generate 68881 code.
-A	Non-ANSI keywords "cdecl" and "pascal" will generate errors.
-B	Write objects (.O) in DRI format, instead of Pure C format. If this switch is on, then the Pure Debugger will not be able to use a source file window because the DRI format does not permit extended debugging information.
-E#	Maximum number of error messages before break.
-F#	Maximum number of warning messages before break.
-H	Force "cdecl"-style function calling. Passes parameters on stack instead of in CPU registers.
-L#	Maximum identifier length.
-M	No string merging. Normally, Pure C checks to see if part of one string can actually be defined as being the tail end of another. An example would be the two strings "nobody" and "body", which could be merged into one string. This option can be disabled if your code needs to modify static string contents without affecting other strings.
-Nname	Output file directory.
-Oname	Output file name.
-P	Use absolute calls (JSR's) instead of PC-relative ones. If you get a linker error which says there is a "16-bit PC-relative overflow", this option must be used.
-S	Standard stackframes, using LINK and UNLINK instructions.
-T	Stack checking. An error message will be produced if insufficient stack space has been reserved.
-Wxxx	Disable (-W-xxx) or enable (-Wxxx) a compiler warning message. Similar to the #pragma warn preprocessor directive. Options:

Adherance to ANSI standards:

-Wdup	Redefinition of XXX is not identical. (default: ON)
-Wret	Both return and return of a value used. (ON)
-Wstr	XXX not part of a structure. (ON)
-Wstu	Structure XXX is not defined. (ON)
-Wsus	Suspicious pointer conversion. (ON)
-Wvoi	Void functions may not return a value. (ON)
-Wzst	Zero length structure. (ON)

Common Warning Messages:

-Waus	XXX is assigned a value which is never used. (ON)
-Wdef	Possible use of XXX before definition. (ON)
-Weff	Code has no effect. (ON)
-Wpar	Parameter XXX is never used. (ON)
-Wpia	Possibly incorrect assignment. (ON)

	-Wrch	Unreachable code. (ON)
	-Wrvl	Function should return a value. (ON)
	Less Common Warnings:	
	-Wamb	Ambiguous operators need parentheses. (OFF)
	-Wamp	Superfluous & with function or array. (OFF)
	-Wnod	No declaration for function XXX. (OFF)
	-Wpro	Call to function with no prototype. (ON)
	-Wstv	Structure passed by value. (OFF)
	-Wuse	XXX declared but never used. (OFF)
	Portability Warnings:	
	-Wapt	Non-portable pointer assignment. (ON)
	-Wcln	Constant is long. (OFF)
	-Wcpt	Non-portable pointer comparison. (ON)
	-Wrng	Constant out of range in comparison. (ON)
	-Wsig	Conversion may lose significant digits. (OFF)
	-Wucp	Mixing pointers to signed and unsigned char. (OFF)
	-Wrpt	Non-portable pointer conversion. (ON)
-X	Generate underbars. All identifier names are preceded with a "_" character. This is mostly for compatibility with Mark Williams C modules.	
-Y	Add debug information for use with the Pure Debugger. This option must be set in order to get a C-source window within the debugger.	

Preprocessor

The only unusual preprocessor directive supported by Pure C is **#pragma**. In this implementation, the only option is "warn", for enabling or disabling certain compiler warnings. The syntax is always one of these:

```
#pragma warn -w(enable all warnings)
#pragma warn -xxx(where "xxx" is a warning type to disable)
#pragma warn +xxx(where "xxx" is a warning type to enable)
```

For a list of the warning types, refer to the -W compiler option.

Pure C defines several preprocessor constants:

• __LINE__	Line number.
• __FILE__	Name of file being compiled.
• __DATE__	File compilation date.
• __TIME__	File compilation time.
• __STDC__	True (1) if the "ANSI keywords only" (-A) option is used.
• __PUREC__	Contains the version number of the compiler. Useful as a check to see if the Pure C compiler is being used.
• __TURBOC__	Same as __PUREC__.
• __TOS__	Always 1.

Assembler Options

The -1, -2, -3, -4, -5, and -8 options can be used to prevent an error message when the assembler encounters an instruction which is not part of the standard 68000 instruction set. Similarly, the -S option prevents errors from being generated when a supervisor mode instruction is encountered.

The -U option forces all undefined symbols to be considered external. This avoids having to use the IMPORT and EXPORT assembler directives.

Linker Options

The stack size for a program depends on many factors, such as the number and size of local variables, the level to which function calls are nested, and the number of parameters passed in function calls. The -T (stack checking) compiler switch can be helpful for determining the correct stack size for a particular program. Note that because Pure C normally passes function arguments in CPU registers, the amount of stack space required by a Pure C program can be significantly less than that required by other compilers.

Options G, L, and Y should always be set if you intend to use the resulting program with the Pure Debugger. The program should be re-linked without these options to produce the final version, as this consumes much less disk space.

The -F option prevents the FastLoad bit from being set in the program's header. This attribute only affects programs which are run by TOS versions 1.4 or newer. This attribute must NOT be set (that is, you should use the –F option) for any program which can be run as a desk accessory, since the operating system can crash when a desk accessory has this attribute set.

The -J option generates a new object file. This can also be used to create a linkable library from several .O files.

The -R and -M options affect operation on the Atari TT computer, which has both fast RAM and normal ST RAM. If your program will be performing raw disk i/o or setting the screen display base to a block of memory it receives by way of the Malloc call, then the -M flag should be set so that the Malloc will return a block of memory in ST RAM rather than Fast RAM.

The segment addresses should normally be left blank. They are intended for linking position-dependent code, such as something that will be burned into ROM.

Project (.PRJ) Files

The syntax for Project files follows. Items in braces {} are optional. A bar | denotes a choice between two options.

```
{ output_file | * }
{ .L [ <linker_options> ] }
{ .C [ <compiler_options> ] }
{ .S [ <assembler_options> ] }
=
<module_name1> { ( <dependent_files> ) }
<module_name2> { ( <dependent_files> ) }
...
```

Each module_name can either be an asterisk "*", to represent the topmost source file window, or one of the following:

```
        <assembler_file> { [ <assembler_options> ] }
        <C_source_file> { [ <compiler_options> ] }
        <object_file>
        <library_file>
        <project_file>
```

The order in which modules appear in the list dictates the order in which they are linked. For this reason, the startup module should always come first and the libraries should usually come last.

This sample project file will compile the files MYACC1.C (which is the topmost window), MYACC2.C, MYACC3.S, and will link the standard and GEM libraries. The compiler switch for 68020 code is enabled for MYACC2.C, and the assembler switch for privileged instructions (-S) is set globally. The file MYACC1.C depends on the file HEADER.H not having changed since the last compilation. The output file is MYACC1.ACC.

```
        *.ACC           ; topmost window name with ACC extension
        .S[-S]          ; assembler options
        =
        PCSTART.O       ; startup module comes first
        * (HEADER.H)    ; compile topmost window MYACC1 if it or
                        ; HEADER.H has changed
        MYACC2.C [-2]   ; compile MYACC2.C with 68020 code generation
        MYACC3.S        ; assemble with options above
        PCSTDLIB.LIB    ; link standard library
        PCGEMLIB.LIB    ; link AES/VDI library
```

The Runtime Startup Modules

Pure C includes several different compiler startup modules, for different purposes. Most likely, you will probably want to use the default, PCSTART.S.

Essentially, these startups all perform the same actions:

- Find out how much memory the program requires and return (Mshrink) unused memory.
- Parse the command line and prepare a list of pointers to the individual parameters.
- Prepare a list of pointers to the environmental variables.
- Determine if the computer has a floating point math coprocessor.
- Call the main portion of the program.
- Clean-up any malloc'd blocks and open files.
- Terminate.

A program's "main" function receives three parameters: argc, argv, and envp. The first two are common to most C compilers. The envp parameter is a pointer to a null-terminated array of pointers to environmental variables. For example, to display a program's environment, you could use:

```
int main( int argc, char *argv[], char *envp[] ) {
        while( *envp ) {
                printf( *envp );
                envp++;
        }
}
```

Please note that since the runtime startups all look at the return value from "main" and return this to the process which executed the program, it is a good practice to always return something, most likely zero. In the case of the sample program above, a line with "return 0;" should be added.

Writing Desk Accessories

Pure C's startup modules all contain a test to see if a particular program is running as a desk accessory or as a regular program. The int "_app", which is defined in AES.H, is zero if the program is running as a desk accessory.

Helpful Hints

Using CFG and PRJ files:

- Whenever you begin a new project, it is a good idea to start by establishing a new configuration (CFG) file. This can be done by selecting the "Load..." option in the "Options" drop-down menu. If you press the New button in the item selector, you will be asked for the name of a new CFG file.
- From this point, you should set the compiler and assembler options to the way you will want them for the particular program you are compiling. You should also create a new project (PRJ) file or select an existing one. Be sure to Save Configuration if you do not have the Autosave option turned on.
- Now, whenever you want to work on the program further, you can simply call up the "Load..." menu entry and give it the name of the CFG file. Another method is to use the desktop's Install Application feature to install PC.PRG for the extension CFG. This way, you simply have to double-click on the CFG file to get PC.PRG to come up with the correct files loaded.

Using warning level 1:

In the Compiler Options dialog, there are three warning levels which can be used:

- 0 Ignore all warning messages
- 1 Produce some warning messages
- 2 Produce all warning messages

Experience has shown that, for most applications, number 1 is probably the best choice. This level corresponds to the ON/OFF defaults listed in the section of this document which describes the -W compiler switch. Level 0 ignores so many warnings that the code which is generated may not be runable if the -H compiler switch is not used.

Pure C English Overview

Pre-defined Data Types

Type	**sizeof**	**Bits**	**Range**
unsigned char	1	8	0 to 255
char	1	8	-128 to 127
enum	2	16	-32768 to 32767
unsigned short	2	16	0 to 65535
short	2	16	-32768 to 32767
unsigned int	2	16	0 to 65535
int	2	16	-32768 to 32767
unsigned long	4	32	0 to 4294967295
long	4	32	-2147483648 to 2147483647
<pointer>	4	32	
Float	4	32	(+-)3.4E-38 to (+-)3.4E+38
Double	10	80	(+-)3.3E-4932 to (+-)1.2E+4932
long double	10	80	(+-)3.3E-4932 to (+-)1.2E+4932
Zeiger	4	32	0 to 4294967295

The format for floating point numbers is IEEE-standard.

The Help Compiler

Pure C includes a help compiler program (HC.TTP) which can be used to create your own help files.

HC accepts the name of a file as its only parameter. This file must be in the following format:

```
<options>
<HLP_file_name>
<source_file1>
<source_file2>
...
```

The options available are:

L	Produce a log file
N	Parse source file, but do not generate help file
T=n	Expand tabs to "n" characters, where 0 < n <= 9. The default is 4.
V	Verbose message output
W	Break on warnings

Here is a sample command file:

```
-LVT=8; log file on, verbose output, tab size=8
USR.HLP ; Help file name
USR.TXT ; Source file
```

The source files for the help compiler contain blocks of separated text. The basic format is:

```
screen( "<first screen name>" )
Help text here
more help text
\end
screen( "<second screen name>" )
Help text
\end
```

The screen names are the index entries which trigger the help text which follows. More than one screen name can also be specified for the same text:

```
screen( "Cat", "Dog", "Fish" )
Household pets
\end
```

Context-sensitive help (which can be accessed with the Help key) can also be generated by using the directives "sensitive" and "capsensitive":

```
screen( "Index entry #1", sensitive( "keyword" ) )
This text will display for "Keyword", "KEYWORD", or even "kEYwoRd"
\end
screen( "Index entry #2", capsensitive( "Key" ) )
This text will only display for the word "Key"
\end
```

Help screens can also be linked by using the sequence \#. The following two help messages are linked together:

```
screen( "First" )
This is some text.
\end
screen( "Second" )
Double-click on the underlined part to go to the \#First\# text.
\end
```

The use of \# depends on the contents of the text between the markers. Another directive, \link, can also be used so that the name of the actual help screen need not appear:

```
screen( "Second" )
\link( "First" ) Double-click here\# to go to the first screen.
\end
```

Both the PC_HELP desk accessory and the on-line help in the Pure C environment look for a file called USR.HLP. This file can contain your own help messages which will be scanned whenever the help system is accessed by either of these two methods.

The Pure Debugger

The majority of the options in the Pure Debugger should be self-evident. However, a few things should be pointed out.

First of all, probably the most useful thing of all is the double-click.It can be used to change breakpoints, file contents, variables, memory, and just about everything else. When the Inspect option is displaying the value of an array or structure, you can even double-click on a portion of the variable to get another window with a full display of its value. Try this with a nested structure to see for yourself.

Setting a simple breakpoint can be accomplished by clicking once with the mouse within the narrow column on the left side of a source file or assembly listing window. Double-clicking will bring up a dialog with more detailed options. Remember that you can use any combination of these options; you can even have a global breakpoint (which will break anywhere in the program) by turning off the "Breakpoint at:" option. The program will execute much more slowly, but this can be very useful nonetheless.

When a program is running, you can get back to the debugger at any time by pressing Alt-Help.

The main difference between a Watched value and an Inspection is that when a pointer is being examined, a Watch tends to change more often. This is because an Inspection deals with indirected values, whereas a Watch deals with the actual value before indirection, and the actual value is more likely to change than the place it points to.

The following reserved names for CPU registers (as well as a program's variable names) can be used as part of an expression for most addresses, even things like the address at which to Dump memory:

Pseudo Variable	Data Type
D0-D7	unsigned long
A0-A7	char *
PC	char *
FP0-FP7	long double
USP	char *
SSP	char *

Remember that if you want the Pure Debugger to use the source code in its displays, you must not only compile C modules with the -Y option, but you must also link using the -G, -L, and -Y options.

One hint: if your program intercepts system vectors, you should always try to make sure that you allow it to finish before quitting the debugger.

Otherwise, very nasty things can happen.

Pure C English Overview

Pure Assembler

The format used by Pure C's assembler is essentially the same which is used by other assemblers, such as Atari's MADMAC. The following directives are implemented. Each can be preceded with a period ("."), though this is not required. Directives which take a "size" can have one of the following values:

Symbol	**type**	**length (bytes)**
.b	byte	1
.w	word	2
.l	long	4
.s	single precision real	4
.x	extended precision real	10
.p	packed BCD	12

Constants can begin with various prefixes:

$	hexadecimal
0x	hexadecimal
0X	hexadecimal
%	binary
@	octal
<none>	decimal

A constant can also contain underscores (_) to separate its digits for clarity. For instance, one million could be written 1_000_000. Floating point numbers can be written in the same formats that a C compiler understands.

String constants begin with ' or ". A string constant must end with the same character it begins with. A constant can even be used in normal instructions:

```
	move.b# 'A', d0		; set d0.b to 65
```

Directives:

expression = value	Assign a value. For instance:ROM_base = $fc0000
*= expression	Set position forward. For instance, to leave a gap of 256 bytes: *= $100
ALIGN [expression]	Fills with null bytes until the next address divisible by "expression" is encountered. The default value for "expression" is 2 (word alignment).
ALINE #expression	Generates a Line A instruction. Ex: ALINE $10; generates opcode $A010
ASCII string[,string...]	A string of characters, without a NUL at the end. Ex: hello: ASCII "Hello there!", "How are you?"
ASCIIL string[,string...]	A string of characters, preceded by a length byte. Ex.: these two are equivalent: ASCIIL "Hello World!" ; and DC.B12 ASCII "Hello World!"

ASCIIZ string[,string...]	A string of characters followed by a NUL byte. Ex.: these two are equivalent: ASCIIZ "Hello World!" ; and ASCII "Hello World!" DC.B 0
BSS [expression] BSS "name"	Enter the BSS segment. All subsequent instructions will become part of this segment, until a DATA or TEXT directive is encountered. "expression" can be the number of a segment, from 0 to 3. A "name" for a segment can also be given. When the linker links the segments, it always goes in order from 0 to 3.
COMM label,expression	Defines a block of "expression" NUL-filled bytes in the BSS segment.
DATA [expression] DATA "name"	Enter the DATA segment. All subsequent instructions will become part of this segment, until a BSS or TEXT directive is encountered. "expression" can be the number of a segment, from 0 to 3. A "name" for a segment can also be given. When the linker links the segments, it always goes in order from 0 to 3.
DC[.size] expression [,expression...]	Defines a constant in the current segment. Each "expression" in the list is "size" bytes long.

DCB[.size] count [,expression...]	Defines "count" repetitions of the remaining expressions. Ex.:
	`dcb.l 2,-2 ; result: $FFFFFFFEFFFFFFFE`
	`dcb.w 2,3,4 ; $0003000400030004`
	`dcb.b 3,"ABC" ; $414243414243414243`

DS[.size] expression	Reserves "expression" NULL bytes. Used primarily in the BSS segment.
ELSE	See IF.
END	End assembly. Text following END is not evaluated.
ENDC	See IF.
ENDIF	Same as ENDC. See IF.
ENDM	End macro definition. See REPT and MACRO.
ENDMOD	Ends a module. See MODULE.
EQU label, expression label EQU expression	This performs the same function as using = or SET.
ERROR "message"	Terminate the assembler with an error message.
EVEN	Perform word-alignment by inserting a NUL byte, if necessary.
EXITM	Exits a macro without processing it any further.
EXPORT label[,label...]	Defines a label as being global. This is necessary for the linker to resolve external references to a label.
FLINE #expression	Generates a Line F instruction. Ex: FLINE $10 ; generates opcode $F010
GLOBL label[,label...]	Defines one or more labels as being imported into the assembly module, or as being exported out of it.

IFcc expression
  <statements>
[ELSE IF
  <statement>]
ENDIF
	Conditional assembly.

The condition code “cc” can be one of the following:

IF exp	exp != 0
IFF exp	exp == 0
IFB arg	macro arg is not supplied
IFNB arg	macro arg is supplied
IF1	first assembler pass
IF2	second pass
IFEQ exp	exp == 0
IFNE exp	exp != 0
IFLE exp	exp <= 0
IFLT exp	exp < 0
IFGE exp	exp >= 0
IFGT exp	exp > 0

IMPORT label[,label...]	Defines a label as being contained in another module. This is necessary for the linker to resolve external references to a label.
INCLUDE "filename" INCLUDE 'filename'	Includes (assembles) a secondary file.
LCOMM label, expression	Reserves "expression" bytes in the BSS segment with the "label".
LIST	All text following a LIST directive appears in the listing file generated during assembly. Use NOLIST to disable the list.
LOCAL label [,label...]	Defines a local label within a macro in the form ____XXXX, where "XXXX" is from 0000 to 9999. For example:

```
macro absolute
      local    end
      tst.w    d0
      bge      end
      neg.w    d0
   end:
endm
absolute; uses label ____0000
absolute; uses label ____0001
```

- MACRO name[.size] [[param1],param2...]
      <statements>
  ENDM — Define a macro with name "name". For example:

```
Macro   Push.size       parm
        move.size       parm,-(sp)
endm
Push.l  d0              ; generates move.l d0,-(sp)
Push    d0              ; generates move d0,-(sp)
```

To substitute a parameter within a macro in a place which is not preceded with a separator, the & can be used:

```
Macro   PushData     RegNumber
        move.l      D&RegNumber, -(sp)     ; move.l Dx, -(sp)
endm
```

MC68000 MC68010 MC68020 MC68030 MC68040 MC68851 MC68851 - MC68881 [expression] MC68881 -	Selects the specific type of code to be generated by assembler
MODULE label	Defines a module with name "label". A module serves as a convenient way of preventing the linker from treating different occurrences of the same label within one source file as the same occurrence. This is similar to the behavior which occurs when multiple assembly source files are used. Note that for library creation, all the individual routines should either be declared as independent MODULEs, or they should be in separate source files. A MODULE should end with ENDMOD.
NOLIST	Turn off the listing feature. See LIST.
OFFSET [expression]	Generate constants which define the number of bytes from the start of the OFFSET block. An OFFSET block is terminated by changing segments with TEXT, DATA, or BSS directives:

```
; Generate offsets for the elements of the C structure:
; struct list
; {
; struct list *next;
; charname[20];
; }
OFFSET
next: ds.l1 ; next gets 0
name: ds.b20; name gets 4
EVEN
TEXT
move.lnext(a0), a0
lea name(a0), a1
```

• ORG expression	Advance position by "expression" bytes. Same as *=.
PAGE [expression]	Set the page length for listings to "expression". If no value is given, a form feed is generated.
PRINT ["message"]	Prints a message within a listing. If no message is given, a newline is generated.
REG label,registerlist	
label REG registerlist	Defines a register list for the MOVEM instruction. Ex.:

```
SavedRegs REG A2-A4/D3
movem.l#SavedRegs, -(sp)
...
movem.l(sp)+, #SavedRegs
```

REPT expression
    <statements>
ENDM
	Repeat statements "expression" times.

SET label, expression	Same as = and EQU.
label SET expression	
TEXT [expression] TEXT "name"	Enter the TEXT segment. All subsequent instructions will become part of this segment, until a DATA or BSS directive is encountered. "expression" can be the number of a segment, from 0 to 3. A "name" for a segment can also be given. When the linker links the segments, it always goes in order from 0 to 3.
SUPER	Select the supervisor instruction set.
TTL "title"	Define the title for a listing file. The name of the source file can be included in "title" by using %f. Ex.:TTL "Listing for source file %f."
USER	Select the user instruction set.
XDEF label[,label...]	Exports labels for use in external modules. See EXPORT.
XREF label[,label...]	Imports labels from external modules. See IMPORT.

The Pure Assembler will normally try to optimize certain instructions. For a list of which instructions are optimized, see pages 189-190 of the Pure Assembler manual.

Assembly Language Considerations: Parameter Passing

The Pure C compiler passes function parameters in CPU registers to improve performance. The data registers D0, D1, and D2 are for passing char's, int's, and long's. The address registers A0 and A1 are for passing pointers. Any parameters which cannot fit into the correct set of registers are passed on the stack. This includes data types such as "double" and all structures.

To determine which parameter is passed in which register, the function should be evaluated from left to right. For instance, say we have the following function prototype:

```
int foo(char d1, char *p1, struct xx *p2, long *p3, int d2,
        GRECT g1, long d3)
```

and this function was being called as follows:

```
foo( 'A', string, xxptr, longptr,val,grect, 43L )
```

In this case, the compiler might generate code which looks something like this:

```
moveq.l #43, D2          ; parameter d3 into D2 register
lea     grect+8(pc),a0  ; get address of end of grect structure
move.l  -(a0), -(a7)    ; push last two elements
move.l  -(a0), -(a7)    ; push first two elements
move    val,D1          ; parameter d2 into D1 register
move.l  longptr, -(a7)  ; goes on stack because A0 and A1 to be used
move.l  xxptr, A1       ; parameter p2 into A1 register
move.l  string, A0      ; parameter p1 into A0 register
move    #'A', D0        ; parameter d1 into D0 register
jsr foo                 ; call foo
```

This type of parameter passing can be defeated in two ways: either with the -H compiler switch, or by declaring the function to be of type "cdecl". For example, to pass all parameters to the function above on the stack, you could use:

```
int cdecl foo( ...
```

Certain system vector routines (like the critical error handler) can be directly replaced by a Pure C function if the "cdecl" type is used in the function's declaration.

Another implicit case when all parameters are passed on the stack is when the ANSI C ellipses operator ("...") is used. One example of this is the printf function:

```
int printf( const char *format, ... )
```

The compiler uses D0-D2/A0-A2 for parameter passing and for temporary storage. This not only means that any assembly module MUST save any other registers it uses, it also means that any routine which calls a routine compiled by Pure C will most likely have these registers destroyed. For this reason, special care must be taken when writing interrupt handlers using Pure C.

Assembly Language Considerations: Return Values

Return values of type char, int, and long are always returned in the D0 CPU register. Pointers are always returned in the A0 register. More complex data types, such as "double" actually return their values using a pointer to a variable of that type which is passed on the stack. For instance, the following function:

```
double donothing(void) {
        return 1.0;
}
```

when called might generate something like this:

```
lea     10(a7), a0     ; a pre-defined storage space on the stack
move.l  a0, -(a7)      ; push it
jsr     donothing
```

The function "donothing" simply takes this pointer and modifies it:

```
donothing:
move.l  4(a7), a0      ; get pointer to return
move.l  #..., (a0)+    ; set the value...
move.l  #..., (a0)+
move    #..., (a0)
rts
```